COMPLETE PYTHON PROGRAMMING

INDEX

1. **Understanding Programming Concepts**
   - 1.1. Introduction to Programming
   - 1.2. Why Python?
   - 1.3. Python's versatility and applications

2. **Setting Up Your Environment**
   - 2.1. Installing Python
   - 2.2. Choosing a text editor or IDE
   - 2.3. Running your first Python program

3. **Variables and Data Types**
   - 3.1. Understanding variables
   - 3.2. Data Types
   - 3.3. Operations and Methods of Data Types
   - 3.4. List Data Type
   - 3.5. Tuple Data Type
   - 3.6. Dictionary Data Type
   - 3.7. Sets Data Type

4. **Control Flow**
   - 4.1. Conditional statements (if, else, elif)
   - 4.2. Loops (for loops, while loops)
   - 4.3. Break and continue statements

5. **Comprehensions of Python Data Types**
   - 5.1. List comprehension
   - 5.2. DIctionary comprehension
   - 5.3. Set comprehension

6. **Functions**
   - 6.1. Defining functions
   - 6.2. Parameters and arguments
   - 6.3. Return statements
   - 6.4. Scope of variables
   - 6.5. Lambda function
   - 6.6. Map function
   - 6.7. Filter function

7. **Modules and Packages**
   - 7.1. Importing modules
   - 7.2. Creating your own modules
   - 7.3. Exploring Python's standard library

8. **File Handling**
   - 8.1. Reading from files
   - 8.2. Writing to files
   - 8.3. Using context managers (with statement)

9. **Object-Oriented Programming (OOP)**
   - 9.1. Introduction to OOP
   - 9.2. Classes and objects
   - 9.3. Encapsulation
   - 9.4. Inheritance
   - 9.5. Polymorphism
   - 9.6. Abstraction

10. **Exception Handling**
    - 10.1. Understanding exceptions
    - 10.2. Try, except, else, and finally blocks
    - 10.3. Raising exceptions

11. Logging In python

12. Regular Expressions

12.1. Pattern matching with regular expressions

12.2. Using the re module

1. UNDERSTANDING PROGRAMMING CONCEPTS

1.1. Introduction To Programming

Programming is the art of instructing computers to perform specific tasks or solve problems using a set of instructions, also known as code. These instructions are written in a programming language, such as Python, Java, or C++, which the computer can understand and execute.

Python was introduced by **Gudio Van Rossum.**

At its core, programming involves breaking down complex problems into smaller, manageable tasks and then devising a series of steps to solve each task. This process requires logical thinking, problem-solving skills, and attention to detail.

Programming languages serve as a medium for expressing these instructions to the computer. Each programming language has its syntax (rules for writing code) and semantics (meaning of the code). Python, for example, is known for its simplicity and readability, making it an excellent choice for beginners and experienced programmers alike.

Understanding programming concepts is essential for anyone interested in computer science, software development, or technology in general. It opens up a world of possibilities, allowing individuals to create software, build websites, automate tasks, analyze data, and much more.

Consider a scenario where you want to create a program that calculates the average temperature for a week based on daily temperature readings. Programming allows you to write code that takes input data (the daily temperatures), processes it (calculates the average), and provides output (the average temperature for the week). This simple example demonstrates how programming can be used to solve real-world problems efficiently.

1.2. Why Python?

Python has gained immense popularity in recent years, becoming one of the most widely used programming languages across various industries. There are several reasons why Python stands out among other programming languages:

Key points for Why Python?

1. **Simplicity:** Python's syntax is straightforward to understand, making it accessible to beginners. Its readability allows developers to write clean and concise code, reducing the time and effort required for development and maintenance.

2. **Versatility:** Python is a versatile language that can be used for a wide range of applications, including web development, data analysis, artificial intelligence, machine learning, automation, scientific computing, and more. Its extensive standard library and third-party packages provide tools and modules for almost any task imaginable.

3. **Community Support:** Python has a vibrant and active community of developers who contribute to its growth and development. This community-driven approach ensures that there are ample resources, tutorials, forums, and libraries available to help programmers

at all skill levels.

4. **Cross-platform Compatibility:** Python is a cross-platform language, meaning that code written in Python can run on various operating systems, including Windows, macOS, and Linux, without any modifications. This flexibility makes it an ideal choice for developing applications that need to run on multiple platforms.

5. **Scalability:** Python is suitable for projects of all sizes, from small scripts to large-scale enterprise applications. Its scalability and robustness make it a preferred choice for startups, tech giants, and everything in between.

6. **Integration:** Python seamlessly integrates with other programming languages and technologies, allowing developers to leverage existing code and infrastructure. It can easily interface with C/C++, Java, and .NET, among others, making it an excellent choice for building complex systems.

Imagine you're a data scientist working for a retail company. You need to analyze customer purchasing patterns to optimize marketing strategies. Python's versatility allows you to use libraries like Pandas for data manipulation, Matplotlib and Seaborn for data visualization, and scikit-learn for machine learning algorithms. With Python, you can efficiently analyze large datasets, derive insights, and make data-driven decisions to drive business growth.

1.3.Python's Versatility and Applications

Imagine you're a software engineer working for a tech startup. Your team is developing a platform that uses machine learning to recommend personalized products to users based on their preferences and browsing history. Python's versatility allows you to build the web application using Django, implement machine learning algorithms for recommendation using scikit-learn, and deploy the application to the cloud using services like AWS or Google Cloud Platform. With Python, you can seamlessly integrate various components of the system and deliver a scalable and user-friendly solution to your customers.

Python's versatility extends across a wide range of domains, making it a preferred choice for various applications. Here are some key areas where Python excels:

1. **Web Development:** Python offers powerful frameworks like Django and Flask for building dynamic and scalable web applications. These frameworks provide robust features for handling web requests, managing databases, and implementing security measures.

2. **Data Science and Machine Learning:** Python is widely used in data science and machine learning due to its rich ecosystem of libraries. Packages like NumPy, Pandas, Matplotlib, and scikit-learn facilitate data manipulation, analysis, visualization, and modeling, empowering data scientists to extract valuable insights from data.

3. **Artificial Intelligence (AI) and Natural Language Processing (NLP):** Python is extensively used in AI and NLP applications. Libraries such as TensorFlow, Keras, and PyTorch enable developers to build sophisticated neural networks and deep learning models for tasks like image recognition, speech recognition, and language translation.

4. **Scientific Computing:** Python is popular among scientists and researchers for scientific computing tasks. Libraries like SciPy and NumPy provide efficient numerical computation capabilities, enabling scientists to solve complex mathematical problems, simulate

physical systems, and analyze experimental data.

5. **Automation and Scripting:** Python's simplicity and ease of use make it ideal for automation and scripting tasks. Whether it's automating repetitive tasks, managing system configurations, or scripting administrative tasks, Python's versatility allows developers to create efficient and scalable solutions.

6. **Desktop GUI Applications:** Python offers frameworks like Tkinter, PyQt, and wxPython for developing cross-platform desktop graphical user interface (GUI) applications. These frameworks allow developers to create intuitive and interactive desktop applications with ease.

2. Setting Up Your Environment

2.1. Installing Python

Installing Python is the first step towards getting started with programming in Python. Python is available for various operating systems, including Windows, macOS, and Linux. Here's a guide on how to install Python on your computer:

01. Windows:

1. Visit the official Python website at python.org.
2. Navigate to the "***Downloads***" section and click on the latest version of Python for Windows.
3. Download the installer executable (.exe) file.
4. Double-click the downloaded file to launch the installer.
5. Check the box that says "***Add Python <version> to PATH***".
6. Click "***Install Now***" to begin the installation process.
7. Once the installation is complete, you can verify by opening a command prompt and typing "***python --version***".

02. macOS:

1. macOS typically comes with Python pre-installed. However, it's recommended to install the latest version using Homebrew or the official Python installer.
2. If using Homebrew, open Terminal and run the command "***$brew install python***".
3. If using the official installer, download the macOS installer from "python.org" and follow the installation instructions.
4. After installation, you can verify by opening Terminal and typing "***$python3 --version***".

03. Linux:

1. Most Linux distributions come with Python pre-installed. You can check the installed version by opening a terminal and typing "***$python --version***" or "***$python3 --version***".
2. If Python is not installed or you need a different version, you can use your distribution's package manager to install it. For example, on Ubuntu, you can use "***$sudo apt-get install python3***".

2.2. Choosing a Text Editor or IDE

When writing Python code, you have a variety of options for text editors and Integrated Development Environments (IDEs). These tools provide features like syntax highlighting, code completion, debugging, and project management to make your coding experience more efficient and productive.

Text Editors:

Text editors are lightweight programs that are often preferred by developers who want simplicity and flexibility.
Some popular text editors for Python programming include:

1. **VS Code (Visual Studio Code):** VS Code is a free, open-source code editor developed by Microsoft. It offers built-in support for Python development, as well as a rich ecosystem of extensions for additional functionality.
2. **Sublime Text**: Sublime Text is a highly customizable text editor with a clean interface and powerful features like multiple selections and split editing.
3. **Atom:** Atom is an open-source text editor developed by GitHub. It's known for its ease of use, customizable interface, and a wide range of community-created packages.

Integrated Development Environments (IDEs):

IDEs are comprehensive software suites that provide all the tools you need for software development in one package.
Some popular IDEs for Python programming include:

1. **Jupyter Notebooks**: Jupyter Notebooks are web-based interactive notebooks that allow you to write and execute Python code, visualize data, and create narrative documents. They are widely used in data science and education.

2. **PyCharm**: PyCharm is a powerful IDE developed by JetBrains specifically for Python development. It offers features like intelligent code completion, code analysis, and integrated version control.
3. **Spyder:** Spyder is an open-source IDE designed for scientific computing and data analysis with Python. It includes features like variable explorer, debugging tools, and support for interactive scientific computing workflows.

Choosing the right text editor or IDE is an important decision for Python programmers. Whether you prefer the simplicity of a text editor or the comprehensive features of an IDE, there are plenty of options available to suit your coding needs and preferences.

2.3. Running Your First Python Program

After installing Python and choosing a text editor or Integrated Development Environment (IDE), you're ready to write and run your first Python program. This exciting step marks the beginning of your journey into Python programming.

Writing Your Python Program:

First, open your chosen text editor or IDE and create a new file. Then, type the following simple Python program:

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

This program consists of a single line of code that prints the message "Hello, world!" to the console. It's a classic example often used to demonstrate the basic syntax of a programming language.

Saving Your Python Program:

Save the file with a "***.py***" extension, such as "***hello.py***". This extension indicates that the file contains Python code. Choose a location on your computer where you can easily access the file.

Running Your Python Program:

Now, it's time to run your Python program. Open a terminal or command prompt, navigate to the directory where you saved your ***hello.py*** file, and type the following command:

```
python hello.py
```

This command tells the Python interpreter to execute the code in the hello.py file. After running the command, you should see the output "Hello, world!" displayed in the terminal.

Key Points:

1. Python programs are saved in files with a .py extension.
2. To run a Python program, open a terminal or command prompt, navigate to the directory containing the program file, and use the ***python*** command followed by the name of the file.
3. The print() function is used to display output in Python programs.

Running your first Python program is an exciting milestone in your journey to learn Python programming. It's a simple yet significant step that introduces you to the process of writing code, saving files, and executing programs.

3. VARIABLES AND DATA TYPES

3.1. Understanding Variables

In programming, variables are used to store and manipulate data. They act as placeholders that can hold different values during the execution of a program. Understanding variables is fundamental to writing effective Python code.

Variables in Python:

In Python, variables are created by assigning a value to a name using the assignment operator **(=)**. Unlike some other programming languages, Python is dynamically typed, meaning you don't need to specify the data type(int, float, bool, ...) of a variable explicitly. Python infers the data type based on the value assigned to the variable.

Rules for Identifiers or Variables:

1. Must begin with a letter (a-z, A-Z) or an underscore (_)
2. Can contain letters, numbers, and underscores
3. Cannot start with a number
4. Cannot use special characters such as !, @, #, $, %, etc.

Examples of Right Variables or Identifiers:

- Name
- Age
- First_name
- _salary
- totalMarks

Examples of Wrong Variables or Identifiers:

- 2ndName (starts with a number)
- my-name (contains a hyphen)
- $price (contains a special character)

```
name_1 = 'Sai' #(Correct format)
1_name = 'Sai' #(Wrong format - raises error)

_place_1 = 'Banglore' #(Correct format)
1_place = 'Banglore'  #(Wrong format - raises error)
```

3.2. Data Type

Python supports several data types that are commonly used in programming. These data types include int, float, string, boolean, lists, tuples, dictionaries, and sets.

1. Integer

Integers are whole numbers without any decimal point. They can be positive, negative, or zero.

Example:

```
Age = 25
Quantity = -10
print(type(Age))  #Output - int
print(type(Quantity))  #Output - int
```

The **print** function is used to print the output.
The **type** function is used to get the data type of a variable.

In this example, age is assigned the value of 25, and quantity is assigned the value of -10. Both age and quantity are **integers**.

2. Float

Floats, or floating-point numbers, are numbers with a decimal point or numbers in scientific notation.

Example:

```
price = 19.99
weight = 2.5
print(type(price))  #Output - float
print(type(weight))  #Output - float
```

In this example, price is assigned the value of 19.99, and weight is assigned the value of 2.5. Both price and weight are **floats**.

3. String

Text data types, also known as strings, are used to represent sequences of characters in Python. Strings are enclosed within either single quotes **(' ')** or double quotes **(" ")**.

Example:

```
name = 'Alice'
address = "123 Main Street"
print(type(name))  #Output - str
print(type(address))  #Output - str
```

In this example, the name is assigned the string 'Alice', and the address is assigned the string "123 Main Street". Both name and address are **strings**.

4. Boolean

The Boolean data type in Python represents two built-in values: **True** and **False**. Booleans are commonly used in programming to evaluate conditions and make decisions based on whether those conditions are true or false.

Note: *The Bool values should always **start with capital letter, T and F** (If you use **'false' / 'true'** raises an error)*

Example:

```
is_student = True
is_adult = False
print(type(is_student))  #Output - bool
print(type(is_adult))  #Output - bool
```

In this example, is_student is assigned the value of True, indicating that a person is a student, while is_adult is assigned the value of False, indicating that a person is not an adult.

Other Data Type Includes:

1. List
2. Tuple
3. DIctionary
4. Set

We'll discuss about these in-detail.

3.3. Operations and Methods of Data Types

The data types in Python language support performing different operations and different methods to execute things in specific formats. We'll discuss these operations and methods of each data type.

1. Integer Data Type

Integers are whole numbers without any decimal point. They support various operations and methods for performing arithmetic calculations and manipulating integer values.

Arithmetic Operations:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Floor Division (//)
- Modulo (%)
- Exponentiation (**)

Example:

```
x = 10
y = 3
addition = x + y            # Output -  13
subtraction = x - y         # Output - 7
multiplication = x * y      # Output - 30
division = x / y            # Output - 3.3333333333333335
floor_division = x // y     # Output - 3
modulo = x % y              # Output - 1
exponentiation = x ** y     # Output - 1000
```

Comparison Operations:

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

Example:

```
x = 10
y = 5
is_equal = x == y               # Output -  False
is_not_equal = x != y           # Output - True
is_greater_than = x > y         # Output - True
is_less_than = x < y            # Output - False
is_greater_or_equal = x >= y    # Output - True
is_less_or_equal = x <= y       # Output - False
```

Methods and Attributes:

Python has built-in methods and attributes that allow you to perform various operations and access information about integer objects. Some common methods and attributes include:

- abs(): Returns the absolute value of an integer.
- bit_length(): Returns the number of bits necessary to represent the integer in binary.
- to_bytes(): Converts the integer to a byte string.
- from_bytes(): Creates an integer from a byte string.
- real: The real part of a complex number (if the integer is a complex number).

Example:

```
x = -10
absolute_value = abs(x)      # Output -  10
bit_length = x.bit_length()  # OUtput - 4
```

2. Float Data Type

Floats represent decimal numbers.

The methods and tributes are similar to the integer data type.

Arithmetic Operations:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulo (%)
- Exponentiation (**)

Example:

```
x = 3.5
y = 2.0
addition = x + y            # Output - 5.5
subtraction = x - y         # Output - 1.5
multiplication = x * y      # Output - 7.0
division = x / y            # Output - 1.75
modulo = x % y              # Output - 1.5
exponentiation = x ** y     # Output - 12.25
```

Comparison Operations:

These comparison operators are the same as integers.

Methods and Attributes:

- is_integer(): Returns True if the float is an integer, otherwise returns False.
- hex(): Returns a hexadecimal representation of the float.
- as_integer_ratio(): Returns a tuple containing the numerator and denominator of the float's exact value.

Example:

```
x = 3.5
is_integer = x.is_integer()                    #Output - False
hex_representation = x.hex()                   #Output - '0x1.c000000000000p+1'
numerator, denominator = x.as_integer_ratio()  #Output - (7, 2)
```

3. String Data Type

Strings are sequences of characters enclosed within either single quotes **(' ')** or double quotes **(" ")**. They support various operations, methods, and attributes for manipulating and formatting textual data.

Common operations:

1. Concatenation: Combining two or more strings using the **"+"** operator.

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

Here in this example, we are concatenating the two strings, and also we are concatenating the comma with a space(in between greeting and name) and an exclamatory mark(at the end of the message).

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

```
separator = "-" * 10
print(separator)
# Output: "----------"
```

In this example, we are printing the separator value(which is a hyphen value) 10 times(where ***** represents multiplication).

3. Indexing: Accessing individual characters or substrings within a string using indexing.

Here access the index position of a variable and print the character at the specified index/position.

```
word = "Python"
first_letter = word[0]  #output - 'P'
last_letter = word[-1]  #output - 'n'
substring = word[2:5]   #output - 'tho'
```

- first_letter - print character at 0th index.
- second_letter - print character at -1(last) index.
- Third_letter - print character from 2nd - 5th index(exclude 2nd(n+1), include 5th). Here colon specifies **to(**starting **to** ending**).**
- We are using the List format to access the indexes of variables to print the characters. Whenever you need to print the character by using the indexing method, you need to use **[ ].**

4. Formatting Strings: Python offers several ways to format strings, including using the format() method and f-strings (formatted string literals).

```
name = "Alice"
age = 30
message_1 = "My name is {} and I am {} years old.".format(name, age)
message_2 = f"My name is {name} and I am {age} years old."
print(message_1)  # Output: "My name is Alice and I am 30 years old."
print(message_2)  # Output: "My name is Alice and I am 30 years old."
```

Here in this example, in message_1, we use curly braces {} as placeholders for the variables and then use the .format() method to specify the values to be inserted into these placeholders.

In message_2, we directly include the variable names within curly braces {}, and Python automatically replaces them with the corresponding values of name and age.

Common Methods of String:

1. lower() and upper(): Converts the string into lowercase and uppercase.

```
text = "Hello, World!"
lowercase_text = text.lower()
uppercase_text = text.upper()  "
print(loercase_text)  #Output - "hello, world!"
print(uppercase_text) #Output - "HELLO, WORLD!
```

2. find(): Find the first occurrence of a substring within the string.

```
text = "Python is a powerful programming language."
index = text.find("powerful")
print(index)  #Output - 11
```

3. split(): Split the string into a list of substrings based on a delimiter.

```
sentence = "Python is fun to learn"
words = sentence.split()
print(words)  #Output - ['Python', 'is', 'fun', 'to', 'learn']
```

4. len(): Returns the length of the string (number of characters).

```
word = "Python"
length = len(word)
print(length)  #Output - 6
```

5. strip(), lstrip(), and rstrip(): These methods remove leading and trailing whitespace characters (spaces, tabs, newline characters) from the string. strip() removes whitespace from both ends, lstrip() removes whitespace from the left end, and rstrip() removes whitespace from the right end.

```
text_1 = "   Python   "
text_2 = "    Programming    "
text_3 = "    Language     "
print(text_1.strip())   #Output - "Python"
print(text_2.lstrip())  #Output - "Programming    "
print(text_3.rstrip())  #Output - "    Language"
```

6. replace(): This method replaces all occurrences of a specified substring with another substring within the string.

```
sentence = "Python is fun to learn"
updated_sentence = sentence.replace("fun", "exciting")
print(updated_sentence) #Output  "Python is exciting to learn"
```

7. count(): This method returns the number of occurrences of a specified substring within the string.

```
sentence = "Python is fun to learn. Python is powerful."
count_py = sentence.count("Python")
print(count_py) #output  2
```

8. startswith() and endswith(): These methods return True if the string starts or ends with the specified prefix or suffix, respectively.

```
sentence = "Python is fun to learn"
print(sentence.startswith("Python"))  #Output - True
print(sentence.endswith("learn"))     #Output - True
```

9. join(): This method joins elements of an iterable (such as a list) into a single string, with the string itself serving as a separator between each element.

```
words = ["Python", "is", "fun"]
sentence = " ".join(words)
print(sentence)  #Output - "Python is fun"
```

10. capitalize(), title(), and swapcase(): These methods modify the case of the string. capitalize() - capitalizes the first character of the string, title() - capitalizes the first character of each word, and swapcase() - swaps the case of each character in the string.

```
text = "hello world"
print(text.capitalize())   #Output - "Hello world"
print(text.title())        #Output - "Hello World"
print(text.swapcase())     #Output - "HELLO WORLD"
```

3.4. List

Lists are one of the most versatile data structures in Python. They are ordered, mutable collections of items, which means you can change their content without changing their identity. Lists can contain items of different data types, including other lists.

1. Creating Lists

You can create a list by placing all the items (elements) inside square brackets [ ], separated by commas.

Example:

```
# Creating a list of integers
numbers = [1, 2, 3, 4, 5]

# Creating a list of strings
fruits = ["apple", "banana", "cherry"]

# Creating a list with mixed data types
mixed_list = [1, "apple", 3.14, True]
```

2. Accessing List Elements:

You can access elements of a list using indexing. Python uses zero-based indexing, so the first element is at index 0.

Example:

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

# Accessing the first element
print(fruits[0])  # Output: apple

# Accessing the last element
print(fruits[-1])  # Output: cherry
```

3. Modifying Lists

Lists are mutable, which means you can change their content. You can add, remove, or modify elements in a list.

Example:

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

# Adding an element using append()
fruits.append("orange")
print(fruits)  # Output: ['apple', 'blueberry', 'cherry', 'orange']

# Removing an element using remove()
fruits.remove("cherry")
print(fruits)  # Output: ['apple', 'blueberry', 'orange']
```

4. List Operations

You can perform various operations on lists, such as concatenation, repetition, and slicing.

Example:

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

# Repetition
repeated_list = list1 * 3
print(repeated_list)  # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]

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

5. List Methods

Python provides several built-in methods to work with lists. Here are some commonly used methods:

1. append(): Adds an element at the end of the list.
2. extend(): Extends the list by appending elements from another list.
3. insert(): Inserts an element at a specified position.
4. remove(): Removes the first occurrence of a specified element.
5. pop(): Removes the element at the specified position (or the last element if index is not provided) and returns it.
6. clear(): Removes all elements from the list.
7. index(): Returns the index of the first occurrence of a specified element.
8. count(): Returns the number of occurrences of a specified element in the list.
9. sort(): Sorts the list in ascending order (or descending with reverse=True).
10. reverse(): Reverses the order of the list.
11. copy(): Returns a shallow copy of the list.

Example:

```
fruits_1 = ["apple", "banana", "cherry"]
fruits_1.append("orange")
print(fruits_1)  # Output: ['apple', 'banana', 'cherry', 'orange']

#########################################################################
fruits_2 = ["apple", "banana"]
more_fruits = ["cherry", "orange"]
fruits_2.extend(more_fruits)
print(fruits_2)  # Output: ['apple', 'banana', 'cherry', 'orange']

#########################################################################
fruits_3 = ["apple", "banana", "cherry"]
fruits_3.insert(1, "orange")
print(fruits_3)  # Output: ['apple', 'orange', 'banana', 'cherry']

#########################################################################
```

```
fruits_4 = ["apple", "banana", "cherry"]
fruits_4.remove("banana")
print(fruits_$)  # Output: ['apple', 'cherry']

###########################################################################
fruits_5 = ["apple", "banana", "cherry"]
removed_fruit = fruits_5.pop(1)
print(removed_fruit)  # Output: banana
print(fruits_5)  # Output: ['apple', 'cherry']

###########################################################################
fruits_6 = ["apple", "banana", "cherry"]
fruits_6.clear()
print(fruits_6)  # Output: []

###########################################################################
fruits_7 = ["apple", "banana", "cherry"]
index = fruits_7.index("banana")
print(index)  # Output: 1

###########################################################################
fruits_8 = ["apple", "banana", "cherry", "banana"]
count = fruits_8.count("banana")
print(count)  # Output: 2

###########################################################################
numbers_9 = [4, 2, 9, 1]
numbers_9.sort()
print(numbers_9)  # Output: [1, 2, 4, 9]

###########################################################################
numbers_10 = [4, 2, 9, 1]
numbers_10.reverse()
print(numbers_10)  # Output: [1, 9, 2, 4]

###########################################################################
fruits_11 = ["apple", "banana", "cherry"]
fruits_copy = fruits_11.copy()
print(fruits_copy)  # Output: ['apple', 'banana', 'cherry']
```

Example:

1. Managing a Shopping List

Let's create a program to manage a shopping list where users can add, remove, and view items.

```
# Initialize an empty shopping list
shopping_list = []

# Add items to the shopping list
shopping_list.append("milk")
shopping_list.append("bread")
shopping_list.append("eggs")

# Remove an item from the shopping list
shopping_list.remove("bread")

# View the shopping list
print("Shopping List:")
for item in shopping_list:
    print(f"- {item}")
```

Output:

```
Shopping List:
- milk
- eggs
```

Explanation:

- Initialize an empty list shopping_list.
- Add items to the shopping list using append().
- Remove an item from the shopping list using remove().
- Print each item in the shopping list using a for loop.

2. Student Grades Management System

Let's create a program to manage student grades, including adding grades, calculating the average grade, and finding the highest and lowest grades.

```
# Initialize an empty list for student grades
grades = []

# Function to add a grade
def add_grade(grade):
    grades.append(grade)

# Function to calculate the average grade
def average_grade():
    return sum(grades) / len(grades)

# Function to find the highest and lowest grades
def grade_summary():
    return max(grades), min(grades)

# Add grades
add_grade(85)
add_grade(92)
add_grade(78)
add_grade(90)
add_grade(88)

# Calculate the average grade
avg = average_grade()
print(f"Average Grade: {avg:.2f}")

# Find the highest and lowest grades
highest, lowest = grade_summary()
print(f"Highest Grade: {highest}")
print(f"Lowest Grade: {lowest}")
```

Output:

```
Average Grade: 86.60
Highest Grade: 92
Lowest Grade: 78
```

Explanation:

- Initialize an empty list grades.
- Define the add_grade function to add a grade to the list.
- Define the average_grade function to calculate the average grade.
- Define the grade_summary function to find the highest and lowest grades.
- Add grades to the list using the add_grade function.
- Calculate the average grade using the average_grade function.
- Print the average grade.
- Find the highest and lowest grades using the grade_summary function.
- Print the highest and lowest grades.

3.5 Tuples

Tuples are ordered collections of items, similar to lists. However, unlike lists, tuples are immutable, which means once they are created, their elements cannot be changed or modified. Tuples are often used to store related pieces of information that should not be changed, such as coordinates or records from a database.

1. Creating Tuples

You can create a tuple by placing all the items (elements) inside parentheses (), separated by commas.

Example:

```
# Creating a tuple of integers
numbers = (1, 2, 3, 4, 5)

# Creating a tuple of strings
fruits = ("apple", "banana", "cherry")

# Creating a tuple with mixed data types
mixed_tuple = (1, "apple", 3.14, True)
```

2. Accessing Tuple Elements

You can access elements of a tuple using indexing. Python uses zero-based indexing, so the first element is at index 0.

Example:

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

# Accessing the first element
print(fruits[0])  # Output: apple

# Accessing the last element
print(fruits[-1])  # Output: cherry
```

3. Modifying Tuples

Since tuples are immutable, you cannot change their content after they are created. However, you can concatenate or slice tuples to create new tuples.

Example:

```
# Concatenating tuples
tuple1 = (1, 2)
tuple2 = (3, 4)
combined_tuple = tuple1 + tuple2
print(combined_tuple)  # Output: (1, 2, 3, 4)

# Slicing tuples
sliced_tuple = combined_tuple[1:3]
print(sliced_tuple)  # Output: (2, 3)
```

4. Tuple Operations

You can perform operations such as concatenation and repetition on tuples.

Example:

```
# Concatenation
tuple1 = (1, 2)
tuple2 = (3, 4)
combined_tuple = tuple1 + tuple2
print(combined_tuple)  # Output: (1, 2, 3, 4)

# Repetition
repeated_tuple = tuple1 * 3
print(repeated_tuple)  # Output: (1, 2, 1, 2, 1, 2)
```

5. Tuple Methods

Tuples have a few built-in methods, but they are limited compared to lists due to their immutability.

1. count(): Returns the number of occurrences of a specified element in the tuple.
2. index(): Returns the index of the first occurrence of a specified element in the tuple.

Example:

```
fruits_1 = ("apple", "banana", "cherry", "banana")
count = fruits_1.count("banana")
print(count)  # Output: 2

#########################################################################
fruits_2 = ("apple", "banana", "cherry")
index = fruits_2.index("banana")
print(index)  # Output: 1
```

Real-World Example:

1. Managing Employee Records:

Let's create a program to manage employee records using tuples. Each employee record will be represented as a tuple containing the employee's ID, name, and department.

```
# List of employee records
employees = [
    (1, "Alice", "HR"),
    (2, "Bob", "Engineering"),
    (3, "Charlie", "Marketing")
]

# Print each employee record
for employee in employees:
    print(f"Employee ID: {employee[0]}, Name: {employee[1]}, Department:
{employee[2]}")
```

Output:

```
Employee ID: 1, Name: Alice, Department: HR
```

```
Employee ID: 2, Name: Bob, Department: Engineering
Employee ID: 3, Name: Charlie, Department: Marketing
```

Explanation:

- Define a list of tuples where each tuple contains an employee's ID, name, and department.
- Iterate over the list of employee records and print each record's details using tuple indexing.

2. Storing and Accessing Geographic Locations

Let's create a program to store geographic locations (latitude and longitude) and find the distance between two locations using the Haversine formula.

```
import math

def haversine(coord1, coord2):
    # Haversine formula to calculate the distance between two coordinates
    R = 6371  # Earth radius in kilometers
    lat1, lon1 = coord1
    lat2, lon2 = coord2
    dlat = math.radians(lat2 - lat1)
    dlon = math.radians(lon2 - lon1)
    a = math.sin(dlat / 2)**2 + math.cos(math.radians(lat1)) *
math.cos(math.radians(lat2)) * math.sin(dlon / 2)**2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
    distance = R * c
    return distance

# Tuple of coordinates for two locations
location1 = (52.5200, 13.4050)  # Berlin
location2 = (48.8566, 2.3522)   # Paris

# Calculate the distance between the two locations
distance = haversine(location1, location2)
print(f"Distance between Berlin and Paris: {distance:.2f} km")
```

Output:

```
Distance between Berlin and Paris: 878.84 km
```

Explanation:

- Define the 'haversine' function to calculate the distance between two geographic coordinates using the Haversine formula.
- Define tuples for the coordinates of Berlin and Paris.
- Calculate the distance between the two locations using the haversine function.
- Print the calculated distance.

3.6. Dictionaries

Dictionaries in Python are unordered collections of items. They store data in key-value pairs, where each key is unique. Dictionaries are mutable, meaning they can be changed after creation. They are highly efficient for retrieving data when the key is known.

1. Creating Dictionaries

You can create a dictionary by placing items inside curly braces {}, separated by commas. Each item is a key-value pair separated by a colon **":"**.

```
# Creating a dictionary
student = {
    "name": "John Doe",
    "age": 21,
    "courses": ["Math", "Computer Science"]
}

print(student)
```

2. Accessing Dictionary Elements

You can access dictionary elements by referring to their keys.

```
student = {
    "name": "John Doe",
    "age": 21,
    "courses": ["Math", "Computer Science"]
}

# Accessing elements
print(student["name"])  # Output: John Doe
print(student["age"])   # Output: 21
print(student["courses"])  # Output: ['Math', 'Computer Science']
```

3. Modifying Dictionaries

Dictionaries are mutable, so you can change their content by adding, modifying, or removing key-value pairs.

```
# Modifying a value
student["age"] = 22

# Adding a new key-value pair
student["graduated"] = False

# Removing a key-value pair
del student["courses"]

print(student)
```

4. Dictionary Methods

Python provides several built-in methods to work with dictionaries. Here are some commonly used methods:

1. get(): Returns the value for a specified key. If the key does not exist, it returns None (or a specified default value).
2. keys(): Returns a view object that displays a list of all the keys in the dictionary.
3. values(): Returns a view object that displays a list of all the values in the dictionary.
4. items(): Returns a view object that displays a list of the dictionary's key-value tuple pairs.
5. update(): Updates the dictionary with elements from another dictionary object or from an iterable of key-value pairs.
6. pop(): Removes the specified key and returns the corresponding value. If the key is not found, it raises a KeyError.
7. clear(): Removes all items from the dictionary.

Example:

```
student_1 = {
```

```
    "name": "John Doe",
    "age": 21
}

age = student_1.get("age")
graduation_year = student_!.get("graduation_year", "Not Available")

print(age)  # Output: 21
print(graduation_year)  # Output: Not Available

#############################################################################
student_2 = {
    "name": "John Doe",
    "age": 21
}

keys = student_2.keys()
print(keys)  # Output: dict_keys(['name', 'age'])

#############################################################################
student_# = {
    "name": "John Doe",
    "age": 21
}

values = student_3.values()
print(values)  # Output: dict_values(['John Doe', 21])

#############################################################################
student_4 = {
    "name": "John Doe",
    "age": 21
}

items = student_4.items()
print(items)  # Output: dict_items([('name', 'John Doe'), ('age', 21)])

#############################################################################
student_5 = {
    "name": "John Doe",
    "age": 21
}
```

```
additional_info = {"graduated": False, "GPA": 3.5}
student_5.update(additional_info)

print(student_5)
# Output: {'name': 'John Doe', 'age': 21, 'graduated': False, 'GPA': 3.5}

##########################################################################
student_6 = {
    "name": "John Doe",
    "age": 21
}

age = student_^.pop("age")
print(age)  # Output: 21
print(student_6)  # Output: {'name': 'John Doe'}

##########################################################################
student_7 = {
    "name": "John Doe",
    "age": 21
}

student_7.clear()
print(student_7)  # Output: {}
```

Real-World Example:

1. Phonebook

Let's create a program to manage a phonebook where you can add, remove, and search for contacts.

```
# Initialize an empty phonebook
phonebook = {}

# Function to add a contact
def add_contact(name, number):
    phonebook[name] = number
```

```
# Function to remove a contact
def remove_contact(name):
    if name in phonebook:
        del phonebook[name]
    else:
        print(f"{name} not found in phonebook")

# Function to search for a contact
def search_contact(name):
    return phonebook.get(name, "Contact not found")

# Adding contacts
add_contact("Alice", "123-456-7890")
add_contact("Bob", "987-654-3210")

# Searching for a contact
print(search_contact("Alice"))  # Output: 123-456-7890
print(search_contact("Charlie"))  # Output: Contact not found

# Removing a contact
remove_contact("Bob")
print(phonebook)  # Output: {'Alice': '123-456-7890'}
```

Output:

```
123-456-7890
Contact not found
{'Alice': '123-456-7890'}
```

Explanation:

- Initialize an empty dictionary phonebook.
- Define the add_contact function to add a contact to the phonebook.
- Define the remove_contact function to remove a contact from the phonebook.
- Define the search_contact function to search for a contact in the phonebook.
- Add contacts using the add_contact function.
- Search for a contact using the search_contact function.
- Remove a contact using the remove_contact function.
- Print the phonebook after removal.

2. Inventory Management System

Let's create a program to manage an inventory system where you can add items, update item quantities, and check the stock of items.

```
# Initialize an empty inventory
inventory = {}

# Function to add an item
def add_item(item, quantity):
    if item in inventory:
        inventory[item] += quantity
    else:
        inventory[item] = quantity

# Function to update item quantity
def update_item(item, quantity):
    if item in inventory:
        inventory[item] = quantity
    else:
        print(f"{item} not found in inventory")

# Function to check stock
def check_stock(item):
    return inventory.get(item, "Item not found")

# Adding items
add_item("apples", 10)
add_item("bananas", 20)
add_item("oranges", 15)

# Updating item quantity
update_item("bananas", 25)

# Checking stock
print(check_stock("apples"))  # Output: 10
print(check_stock("bananas"))  # Output: 25
print(check_stock("grapes"))  # Output: Item not found

# Displaying inventory
print(inventory)  # Output: {'apples': 10, 'bananas': 25, 'oranges': 15}
```

Output:

```
10
25
Item not found
{'apples': 10, 'bananas': 25, 'oranges': 15}
```

Explanation:

- Initialize an empty dictionary inventory.
- Define the add_item function to add an item to the inventory. If the item already exists, increase its quantity.
- Define the update_item function to update the quantity of an item in the inventory.
- Define the check_stock function to check the stock of an item in the inventory.
- Add items to the inventory using the add_item function.
- Update the quantity of an item using the update_item function.
- Check the stock of items using the check_stock function.
- Print the current inventory.

3.7. Sets

Sets in Python are unordered collections of unique elements. They are mutable, meaning you can add or remove elements, but they do not allow duplicate values. Sets are particularly useful for membership testing and eliminating duplicate entries.

1. Creating Sets

You can create a set by placing elements inside curly braces {} or by using the set() function.

Example:

```
# Creating a set using curly braces
fruits = {"apple", "banana", "cherry"}
print(fruits)  # Output: {'apple', 'banana', 'cherry'}

# Creating a set using the set() function
numbers = set([1, 2, 3, 4, 4, 5])
print(numbers)  # Output: {1, 2, 3, 4, 5}
```

2. Accessing Set Elements

You cannot access items in a set by referring to an index since sets are unordered. However, you can loop through the set items using a for loop.

Example:

```
fruits = {"apple", "banana", "cherry"}

# Looping through set elements
for fruit in fruits:
    print(fruit)
```

3. Modifying Sets

Sets are mutable, so you can add or remove items using methods like add(), remove(), discard(), and pop().

- **Adding Items**

Example:

```
fruits = {"apple", "banana"}

# Adding an item
fruits.add("cherry")
print(fruits)  # Output: {'apple', 'banana', 'cherry'}
```

- **Removing Items**

Example:

```
fruits = {"apple", "banana", "cherry"}

# Removing an item using remove()
fruits.remove("banana")
print(fruits)  # Output: {'apple', 'cherry'}

# Removing an item using discard()
fruits.discard("apple")
print(fruits)  # Output: {'cherry'}

# Removing an item using pop()
removed_item = fruits.pop()
print(removed_item)  # Output: cherry
print(fruits)  # Output: set()
```

Sets support various operations that are useful for mathematical computations, such as union, intersection, difference, and symmetric difference.

1. Union: The union of two sets is a set containing all elements from both sets.
2. Intersection: The intersection of two sets is a set containing only the elements that are present in both sets.
3. Difference: The difference of two sets is a set containing elements that are in the first set but not in the second set.
4. Symmetric Difference: The symmetric difference of two sets is a set containing elements that are in either of the sets, but not in both.

Example:

```
A = {1, 2, 3}
B = {3, 4, 5}

union_set = A.union(B)
print(union_set)  # Output: {1, 2, 3, 4, 5}

########################################################################
A = {1, 2, 3}
B = {3, 4, 5}

intersection_set = A.intersection(B)
print(intersection_set)  # Output: {3}

########################################################################
A = {1, 2, 3}
B = {3, 4, 5}

difference_set = A.difference(B)
print(difference_set)  # Output: {1, 2}

########################################################################
A = {1, 2, 3}
B = {3, 4, 5}

symmetric_difference_set = A.symmetric_difference(B)
print(symmetric_difference_set)  # Output: {1, 2, 4, 5}
```

Real-World Example:

1. Unique Visitors to a Website

Let's create a program to track unique visitors to a website using sets.

```
# Initialize a set for unique visitors
unique_visitors = set()

# Function to add a visitor
def add_visitor(visitor_id):
    unique_visitors.add(visitor_id)
```

```
# Adding visitors
add_visitor("visitor_1")
add_visitor("visitor_2")
add_visitor("visitor_1")  # Duplicate visitor

# Display unique visitors
print("Unique Visitors:", unique_visitors)
```

Output:

```
Unique Visitors: {'visitor_1', 'visitor_2'}
```

Explanation:

- Initialize an empty set unique_visitors.
- Define the add_visitor function to add a visitor to the set.
- Add visitors using the add_visitor function, including a duplicate visitor.
- Print the set of unique visitors.

2. Analyzing Social Media Tags

Let's create a program to analyze social media tags and find common and unique tags between two sets of posts.

```
# Tags from two social media posts
post1_tags = {"python", "coding", "AI", "data"}
post2_tags = {"AI", "machine learning", "coding", "development"}

# Finding common tags (intersection)
common_tags = post1_tags.intersection(post2_tags)
print("Common Tags:", common_tags)

# Finding unique tags in post1 (difference)
unique_to_post1 = post1_tags.difference(post2_tags)
print("Unique to Post 1:", unique_to_post1)

# Finding unique tags in post2 (difference)
unique_to_post2 = post2_tags.difference(post1_tags)
print("Unique to Post 2:", unique_to_post2)
```

```
# Finding tags that are in either post but not both (symmetric difference)
unique_tags = post1_tags.symmetric_difference(post2_tags)
print("Unique Tags in Either Post but not Both:", unique_tags)
```

Output:

```
Common Tags: {'coding', 'AI'}
Unique to Post 1: {'python', 'data'}
Unique to Post 2: {'development', 'machine learning'}
Unique Tags in Either Post but not Both: {'machine learning', 'python',
'development', 'data'}
```

Explanation:

- Define sets of tags for two social media posts.
- Find common tags using the intersection() method.
- Print the common tags.
- Find tags unique to the first post using the difference() method.
- Print tags unique to the first post.
- Find tags unique to the second post using the difference() method.
- Print tags unique to the second post.
- Find tags that are in either post but not both using the symmetric_difference() method.
- Print the unique tags in either post but not both.

4. CONTROL FLOW

Control flow in programming refers to the order in which individual statements, instructions, or function calls are executed or evaluated. In Python, control flow can be managed using conditional statements and loops. These constructs allow you to execute certain blocks of code based on specific conditions or to repeat blocks of code multiple times.

Key Concepts of Control Flow:

1. Conditional Statements:

- If Statements: Executes a block of code if a specified condition is true.
- Else Statements: Executes a block of code if the condition in the if statement is false.
- Elif Statements: Stands for "else if" and allows you to check multiple expressions for true and execute a block of code as soon as one of the conditions is true.

2. Loops:

- For Loops: Iterates over a sequence (such as a list, tuple, dictionary, set, or string) and executes a block of code for each item in the sequence.
- While Loops: Repeats a block of code as long as a specified condition is true.

3. Control Flow Modifiers:

- Break Statements: Terminates the loop and transfers execution to the statement immediately following the loop.
- Continue Statements: Skips the rest of the code inside the loop for the current iteration and moves to the next iteration of the loop.

4.1. Conditional statements (if, else, elif)

Conditional statements are used to perform different actions based on different conditions. They allow you to control the flow of your program by executing specific blocks of code depending on whether a condition is true or false. The primary conditional statements in Python are if, else, and elif.

Explanation

if Statement: Executes a code block if the specified condition is true.
else Statement: Executes a code block if the if condition is false.
elif Statement: Stands for "else if", allows you to check multiple conditions.

Syntax

```
if condition:
    # code block
elif another_condition:
    # Another code block
else:
    # Another code block
```

Examples

Example 1: Basic if Statement

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

This is a basic if statement that checks if the age is 18 or older. Now, let's see how we can integrate the else statement.

Example 2: if and else Statement

```
age = 16
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")  ##Output: You are a minor.
```

In this example, we have added an else statement to handle the case where the condition in the if statement is false. Next, let's explore how to use the elif statement.

Example 3: if, elif, and else Statement

```
age = 20
if age < 13:
    print("You are a child.")
elif 13 <= age < 18:
    print("You are a teenager.")
else:
    print("You are an adult.")  ##utput: You are an adult.
```

This example introduces the elif statement, which allows us to check multiple conditions. Now, let's see how to combine multiple conditions using logical operators.

Example 4: Combining Conditions

```
age = 25
has_ticket = True

if age >= 18 and has_ticket:
    print("You can enter the concert.")
else:
    print("You cannot enter the concert.") ##Output: You can enter the
concert.
```

Here, we combine two conditions using the and operator. Both conditions must be true for the if block to execute. Finally, let's look at how to nest conditions for more complex logic.

Example 5: Nested Conditions

```
age = 25
has_ticket = True
is_vip = False

if age >= 18:
    if has_ticket:
        if is_vip:
            print("Welcome to the VIP section.")
        else:
            print("Welcome to the concert.")
```

```
    else:
        print("You need a ticket to enter.")
else:
    print("You are too young to enter.") ## Output: Welcome to the concert.
```

In this example, we nest conditions to create a more complex decision-making structure.

Real-world example:

1. Grading System:

```
score = 85

if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
    grade = 'D'
else:
    grade = 'F'

print(f"Your grade is: {grade}")
```

Output:

```
Your grade is: B
```

Explanation:

- Assign a value to the variable score.
- Check if the score is 90 or above. If true, assign 'A' to grade.
- If the first condition is false, check if the score is 80 or above. If true, assign 'B' to grade.
- If the previous conditions are false, check if the score is 70 or above. If true, assign 'C' to grade.
- If the previous conditions are false, check if the score is 60 or above. If true, assign 'D' to grade.

- If none of the conditions are met, assign 'F' to grade.
- Print the final grade.

2. Loan Eligibility:

```
age = 30
income = 50000
credit_score = 750

if age >= 18:
    if income >= 30000:
        if credit_score >= 700:
            print("Eligible for loan")
        else:
            print("Not eligible due to credit score")
    else:
        print("Not eligible due to income")
else:
    print("Not eligible due to age")
```

Output:

```
Eligible for loan
```

Explanation:

- Assign values to the variables age, income, and credit_score.
- Check if the age is 18 or above.
    - If true, check if the income is 30,000 or above.
        - If true, check if the credit score is 700 or above.
            - If true, print "Eligible for loan".
            - If false, print "Not eligible due to credit score".
        - If the income condition is false, print "Not eligible due to income".
    - If the age condition is false, print "Not eligible due to age".

4.2. Loops (for loops, while loops)

Loops are used to execute a block of code repeatedly as long as a certain condition is met. Python supports two main types of loops: for loops and while loops. for loops are typically used when the number of iterations is known beforehand, while while loops are used when the number of iterations is not predetermined.

1. **for Loop:** Iterates over a sequence (e.g., list, tuple, dictionary, set, or string) and executes a block of code for each element in the sequence.
2. **while Loop:** Repeatedly executes a block of code as long as a specified condition is true.

"for" Loop Syntax:

```
for variable in sequence:
    # code block
```

"while" Loop Syntax:

```
while condition:
    # code block
```

Examples

Example 1: Basic for Loop

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

Output:

```
apple
banana
cherry
```

This is a basic for loop that iterates over a list of fruits and prints each one.

Example 2: for Loop with "range"

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

Output:

```
0
1
2
3
4
```

Here, the for loop uses the range function to iterate from 0 to 4. Next, let's see how we can use a while loop.

Example 3: Basic while Loop

```
count = 0
while count < 5:
    print(count)
    count += 1  ## (count = count + 1)
```

Output:

```
0
1
2
3
4
```

This is a basic while loop that prints the count from 0 to 4. The loop continues as long as the condition count < 5 is true.

Example 4: Using break in a Loop

```
for i in range(10):
    if i == 5:
        break
```

```
    print(i)
```

Output:

```
0
1
2
3
4
```

The break statement terminates the loop when i equals 5.

Example 5: Using "continue" in a Loop

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

Output:

```
1
3
5
7
9
```

The continue statement skips the current iteration of the loop when i is an even number.

Example 6: Nested Loops

```
for i in range(3):
    for j in range(2):
        print(f"i={i}, j={j}")
```

Output:

```
i=0, j=0
i=0, j=1
```

```
i=1, j=0
i=1, j=1
i=2, j=0
i=2, j=1
```

Nested loops allow you to iterate over multiple sequences simultaneously. In this example, the outer loop runs 3 times, and for each iteration of the outer loop, the inner loop runs 2 times.

Real-world Example:

1. Summing Numbers in a List

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

for number in numbers:
    total += number

print(f"The total sum is: {total}")
```

Output:

```
The total sum is: 15
```

Explanation:

- Create a list of numbers.
- Initialize the variable total to 0.
- Iterate over each number in the list.
    - Add the current number to the total.
- Print the final total sum.

2. Finding Prime Numbers

```
n = 20

for num in range(2, n + 1):
    prime = True
    for i in range(2, int(num ** 0.5) + 1):
```

```
        if num % i == 0:
            prime = False
            break
    if prime:
        print(num)
```

Output:

```
2
3
5
7
11
13
17
19
```

Explanation:

- Set the variable n to 20.
- Iterate over each number from 2 to n.
    - Assume the current number is prime.
    - Iterate over each number from 2 to the square root of the current number.
        - If the current number is divisible by any number in this range, it is not prime.
            - Set prime to False and break the inner loop.
    - If the number is still assumed to be prime, print it.

4.3. Break and Continue Statements

The break and continue statements are used to alter the flow of loops. The break statement is used to exit the loop prematurely, while the continue statement skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

1. **break Statement:** Terminates the loop and transfers execution to the statement immediately following the loop.
2. **continue Statement:** Skips the rest of the code inside the loop for the current iteration and jumps to the next iteration.

Syntax

"break" Statement Syntax:

```
for variable in sequence:
    if condition:
        break
    # code block
```

"continue" Statement Syntax:

```
for variable in sequence:
    if condition:
        continue
    # code block
```

Examples

Example 1: Basic break Statement in a "for" Loop:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Output:

```
0
1
2
3
4
```

In this example, the loop terminates when i equals 5, so the numbers 0 through 4 are printed.

Example 2: Basic continue Statement in a for Loop

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

Output:

```
1
3
5
7
9
```

In this example, the continue statement skips the even numbers, so only the odd numbers are printed.

Example 3: Basic break Statement in a while Loop

```
count = 0
while count < 10:
    print(count)
    count += 1
    if count == 5:
        break
```

Output:

```
0
1
2
3
4
```

In this example, the loop terminates when count equals 5, so the numbers 0 through 4 are printed.

Example 4: Basic continue Statement in a while Loop

```
count = 0
while count < 10:
    count += 1
    if count % 2 == 0:
        continue
    print(count)
```

Output:

```
1
3
5
7
9
```

In this example, the continue statement skips the even numbers, so only the odd numbers are printed.

Real-World Example:

1. Finding First Sold-Out Product:

```
products = [
    {"name": "Laptop", "stock": 10},
    {"name": "Phone", "stock": 5},
```

```
    {"name": "Tablet", "stock": 0},
    {"name": "Monitor", "stock": 8}
]

for product in products:
    if product["stock"] == 0:
        print(f"The first sold-out product is: {product['name']}")
        break
```

Output:

```
The first sold-out product is: Tablet
```

Explanation:

- Create a list of dictionaries representing products with their stock levels.
- Iterate over each product in the list.
- Check if the current product's stock is 0.
- If true, print the name of the first sold-out product.
- Terminate the loop after finding the first sold-out product.

2. Filtering Out Unavailable Flights

```
flights = [
    {"flight": "AA123", "status": "On Time"},
    {"flight": "BA456", "status": "Cancelled"},
    {"flight": "CA789", "status": "Delayed"},
    {"flight": "DA012", "status": "On Time"}
]

for flight in flights:
    if flight["status"] != "On Time":
        continue
    print(f"Flight {flight['flight']} is available for booking.")
```

Output:

```
Flight AA123 is available for booking.
Flight DA012 is available for booking.
```

Explanation:

- Create a list of dictionaries representing flights with their statuses.
- Iterate over each flight in the list.
- Check if the current flight's status is not "On Time".
- If true, skip the current iteration.
- Print the flight information if the status is "On Time".

- Control flow statements are essential in programming for managing the execution of code based on different conditions and scenarios. The if, else, and elif statements allow for conditional execution, enabling your programs to make decisions.
- Loops, such as for and while, enable repetitive execution of code blocks, allowing for efficient handling of tasks that involve iteration.
- The break and continue statements provide additional control over loop execution, allowing for early termination or skipping of specific iterations based on certain conditions.

5. Comprehensions of Python Data Types

Comprehensions provide a concise way to create and manage collections in Python. They are a syntactic construct that allows for the generation of lists, sets, dictionaries, and tuples from existing iterables in a more readable and compact form. The general syntax involves a single line of code that can include conditions and nested loops.

The primary types of comprehensions in Python are:

1. List Comprehension
2. Set Comprehension
3. Dictionary Comprehension
4. Tuple Comprehension

Comprehensions make code more Pythonic by reducing the need for explicit loops and temporary variables, making it easier to read and maintain.

5.1. List Comprehension

List comprehensions are the most commonly used comprehensions in Python. They provide a concise way to generate lists by embedding a for loop and an optional condition inside a pair of square brackets.

Syntax:

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

- **expression:** The value or operation to apply to each item in the iterable.
- **item:** The variable representing the current item in the iterable.
- **iterable:** The collection of items being iterated over.
- **condition:** (Optional) A filter that determines whether the item should be included in the new list.

Example:

Let's create a list of squares of numbers from 0 to 9.

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

Explanation:

- *x**2* is the expression that squares the number.
- *for x in range(10)* is the loop that iterates over numbers 0 to 9.

Example with Condition:

Now, let's create a list of squares for only even numbers from 0 to 9.

```
# List of squares of even numbers from 0 to 9
even_squares = [x**2 for x in range(10) if x % 2 == 0]
print(even_squares) #Output: [0, 4, 16, 36, 64]
```

Explanation:

- The condition if *x % 2 == 0* filters out only even numbers.
- *x**2* squares the even numbers.

Nested List Comprehension:

1. You can also use nested loops in list comprehensions to create lists from multi-dimensional data structures.

```
# Flatten a 2D list into a 1D list
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat_list = [item for sublist in matrix for item in sublist]
print(flat_list) #Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Explanation:

- for sublist in matrix iterates over each sublist in the 2D list.
- for item in sublist iterates over each item in the current sublist.
- item collects the items into the new flat list.

2. In this example, we'll flatten a list of lists where each inner list represents a sentence split into words.

```
# List of sentences
sentences = ["Hello world", "Python is awesome", "Nested list comprehension
example"]

# Flatten the list of sentences into a list of words
words = [word for sentence in sentences for word in sentence.split()]
print(words)
#Output: ['Hello', 'world', 'Python', 'is', 'awesome', 'Nested', 'list',
'comprehension', 'example']
```

Explanation:

- for sentence in sentences iterates over each sentence in the list sentences.
- for word in sentence.split() iterates over each word in the current sentence, obtained by splitting the sentence using .split() method.

- word collects each word into the words list.

Problem Statement:

You have a list of strings containing both words and numbers. Write a Python program to create a new list that includes only the words (strings without any digits).

Solution:

```
# Original list with words and numbers
mixed_list = ['apple', '123', 'banana', '456', 'cherry']

# List comprehension to filter out strings containing digits
word_list = [item for item in mixed_list if not any(char.isdigit() for char
in item)]
print("Filtered Word List:", word_list)
#Output:
Filtered Word List: ['apple', 'banana', 'cherry']
```

Explanation:

- any(char.isdigit() for char in item): Checks if any character in item is a digit.
- if not any(...): Filters out items that contain digits.
- item collects the words into the word_list.

Benefits of List Comprehensions:

1. Conciseness: They allow for writing complex list-generating logic in a single line.
2. Readability: They can make the code more readable by reducing boilerplate code.
3. Performance: They are often faster than traditional for-loops for creating lists because they are optimized for the specific task.

5.2. Dictionary Comprehension

Dictionary comprehensions allow you to create dictionaries using a concise and expressive syntax. They are similar to list comprehensions but produce dictionaries instead of lists and uses curly braces {} instead of square brackets [].

Syntax:

```
{key_expression: value_expression for item in iterable if condition}
```

- key_expression: Expression to generate keys of the dictionary.
- value_expression: Expression to generate values of the dictionary.
- item: Represents each item in the iterable.
- iterable: Iterable like a list, tuple, or range.
- if condition (optional): Condition to filter items.

Example:

1. Creating a Dictionary of Squares

Let's create a dictionary that maps numbers to their squares using dictionary comprehension.

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

Explanation:

- x: x ** 2 for x in range(1, 6): Generates key-value pairs where each key is a number from 1 to 5, and the corresponding value is its square.

2. Filtering Odd Numbers

Let's create a dictionary where keys are even numbers and values are their squares from 1 to 10 using dictionary comprehension with a conditional filter.

```
# Dictionary comprehension with conditional filter
even_squares_dict = {x: x ** 2 for x in range(1, 11) if x % 2 == 0}
print(even_squares_dict)
#Output: {2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

Explanation:

- x: x ** 2 for x in range(1, 11) if x % 2 == 0: Generates key-value pairs where each key is an even number from 1 to 10, and the corresponding value is its square. The condition if x % 2 == 0 filters out odd numbers.

Nested Dictionary Comprehension:

1. Person Details

Let's create a nested dictionary for storing details about different persons, where each person has a name and a dictionary of attributes such as age and occupation.

```
# List of person details
persons = [
    {'name': 'Alice', 'age': 30, 'occupation': 'Engineer'},
    {'name': 'Bob', 'age': 25, 'occupation': 'Teacher'},
    {'name': 'Charlie', 'age': 35, 'occupation': 'Doctor'}
]

# Nested dictionary comprehension to create a dictionary of persons
persons_dict = {person['name']: {'age': person['age'], 'occupation':
person['occupation']} for person in persons}
print(persons_dict)

#Output:
{'Alice': {'age': 30, 'occupation': 'Engineer'}, 'Bob': {'age': 25,
'occupation': 'Teacher'}, 'Charlie': {'age': 35, 'occupation': 'Doctor'}}
```

Explanation:

- {person['name']: {'age': person['age'], 'occupation': person['occupation']} for person in persons}: Generates a nested dictionary where each person's name is the key.
    - For each person in persons, a nested dictionary {age: person['age'], occupation: person['occupation']} is created with age and occupation details.

2. Store Inventory

Let's create a nested dictionary to represent a store's inventory, where each item has attributes like price and quantity.

```
# List of tuples representing store inventory (item, price, quantity)
inventory = [
    ('apple', 1.2, 30),
    ('banana', 0.5, 50),
    ('orange', 1.0, 20),
    ('grape', 2.5, 15),
]

# Nested dictionary comprehension to store inventory details
store_inventory = {item: {'price': price, 'quantity': qty} for item, price,
qty in inventory}

# Print the store inventory
for item, details in store_inventory.items():
    print(f"Item: {item}, Price: ${details['price']}, Quantity:
{details['quantity']}")

#Output:
Item: apple, Price: $1.2, Quantity: 30
Item: banana, Price: $0.5, Quantity: 50
Item: orange, Price: $1.0, Quantity: 20
Item: grape, Price: $2.5, Quantity: 15
```

Explanation:

- inventory: A list of tuples where each tuple represents an item, its price, and its quantity.
- store_inventory: A nested dictionary comprehension that maps each item to a dictionary with price and quantity attributes.

Problem Statement:

You are tasked with creating a program for a bank that manages customers' savings and checking accounts. Each customer can have multiple accounts, and each account has attributes like balance and account type (savings or checking). The program should use nested dictionary comprehensions to store and display the details of each customer's accounts.

Solution:

Here's how you can implement this using nested dictionary comprehensions:

```
# List of tuples representing customer accounts (customer_name,
account_type, balance)
accounts = [
    ('Alice', 'savings', 1500.50),
    ('Alice', 'checking', 1200.00),
    ('Bob', 'savings', 2500.75),
    ('Bob', 'checking', 1800.00),
    ('Charlie', 'savings', 3200.00),
    ('Charlie', 'checking', 1100.50),
]

# Nested dictionary comprehension to store account details
bank_customers = {
    customer: {
        account_type: balance for _, account_type, balance in accounts if
customer == _
    }
    for customer in {customer for customer, _, _ in accounts}
}

# Print the bank customers' account details
for customer, accounts in bank_customers.items():
    print(f"Customer: {customer}")
    for account_type, balance in accounts.items():
        print(f"  {account_type.capitalize()} Account: ${balance}")

#output:
Customer: Alice
```

```
  Savings Account: $1500.5
  Checking Account: $1200.0
Customer: Bob
  Savings Account: $2500.75
  Checking Account: $1800.0
Customer: Charlie
  Savings Account: $3200.0
  Checking Account: $1100.5
```

Explanation:

1. Input Data Structure:

- A list of tuples named accounts is defined, where each tuple represents an account. Each tuple contains:
- customer_name: The name of the customer.
- account_type: The type of the account (savings or checking).
- balance: The balance in the account.

2. Nested Dictionary Comprehension:

- The outer comprehension iterates over a set of unique customer names.
- For each customer, an inner comprehension constructs a dictionary where the keys are account types, and the values are balances. This inner dictionary is formed by filtering the accounts list to include only entries belonging to the current customer.

3. Printing the Output:

- The code then iterates over the bank_customers dictionary and prints each customer's name followed by their account details.

5.3. Set Comprehension

Set comprehensions are similar to list comprehensions and dictionary comprehensions but are used to create sets. Sets are collections of unique elements, so set comprehensions automatically handle duplicate values. They are useful when you need to eliminate duplicates and want a collection of unique items.

Syntax:

```
{expression for item in iterable if condition}
```

- expression: The value to be included in the set.
- item: The variable representing each element in the iterable.
- iterable: The collection of items to iterate over.
- condition: An optional condition to filter the items.

Example

1. Filtering Unique Words from a Sentence

```
sentence = "the quick brown fox jumps over the lazy dog"
unique_words = {word for word in sentence.split()}
print(unique_words)

#output:
{'quick', 'brown', 'dog', 'the', 'over', 'jumps', 'lazy', 'fox'}
```

Explanation:

- sentence.split() splits the sentence into a list of words.
- The set comprehension collects these words into a set named unique_words, automatically removing duplicates.

2. Extracting Unique Email Domains

```
emails = ["alice@example.com", "bob@gmail.com", "charlie@example.com",
"dave@yahoo.com", "eve@gmail.com"]
```

```
unique_domains = {email.split('@')[1] for email in emails}
print(unique_domains)

#Output:
{'example.com', 'gmail.com', 'yahoo.com'}
```

Explanation:

- The comprehension iterates over the list of emails.
- email.split('@')[1] extracts the domain part of each email address.
- The set comprehension collects these domains into a set named unique_domains, automatically removing duplicates.

3. Find Common Divisors

```
# List of numbers
numbers = [12, 18, 24]

# Function to find divisors of a number
def find_divisors(n):
    return {i for i in range(1, n + 1) if n % i == 0}

# Set comprehension to find common divisors
common_divisors = set.intersection(*(find_divisors(num) for num in
numbers))
print(common_divisors)

#Output:
{1, 2, 3, 6}
```

Explanation:

- find_divisors(n): This function returns a set of all divisors of n.
- Nested Set Comprehension:
- common_divisors = set.intersection(*(find_divisors(num) for num in numbers))
    - This part generates a set of divisors for each number in the list numbers.
    - set.intersection(*sets) finds the common elements in all these sets, giving the common divisors of the numbers.

Problem Statement:

You are given a list of student names in an education institute's database. The names have varying capitalizations and extra whitespace. Your task is to clean up this list by normalizing the student names to lowercase and stripping any leading or trailing whitespace, then storing the unique student names in a set.

Solution:

```
# List of student names with varying capitalizations and whitespace
students = ["  John Doe", "jane smith ", "Alice Johnson  ", "john doe",
"JANE SMITH", " Alice johnson "]

# Set comprehension to clean and store unique student names
cleaned_students = {student.strip().lower() for student in students}
print(cleaned_students)

#Output:
{'john doe', 'alice johnson', 'jane smith'}
```

Explanation:

- student.strip() removes leading and trailing whitespace from each student name.
- student.lower() converts each student name to lowercase.
- The set comprehension collects these cleaned student names into a set named cleaned_students, automatically removing duplicates.

6. Functions

6.1. Introduction to Functions

A function is a block of organized, reusable code that performs a specific task. Functions help to break our program into smaller and modular chunks, making it easier to understand, maintain, and debug. They allow you to reuse code by calling the function whenever needed.

Why Use Functions?

1. Code Reusability: Functions allow you to write a piece of code once and use it multiple times without repeating it.
2. Modularity: Functions enable you to break down complex problems into smaller, more manageable parts.
3. Maintainability: Functions make it easier to update and maintain code since changes in the function definition are reflected wherever the function is used.
4. Readability: Functions help to improve the readability of the code by abstracting away complex logic into simple function calls.

Types of Functions

1. Built-in Functions: Functions that are already defined in Python, such as print(), len(), max(), etc.
2. User-defined Functions: Functions that you define yourself to perform specific tasks.

Syntax:

```
def function_name(parameters):
    """
    Docstring (optional): Describes the function.
    """
    # Function body
    statement(s)
    return value (optional)
```

- def: Keyword used to define a function.
- function_name: Name of the function.
- parameters: A list of parameters (arguments) that the function can accept. Parameters are optional.
- ":": Colon symbol that denotes the start of the function body.
- Docstring: (Optional) A short description of what the function does.
- statement(s): The block of code that performs the task. This is also known as the function body.
- return: (Optional) Keyword used to return a value from the function.

Example:

Let's look at a simple example of defining and calling a function.

```
# Defining a function
def greet(name):
    """
    This function greets the person with the provided name.
    """
    print(f"Hello, {name}!")

# Calling the function
greet("Alice")  # Output: Hello, Alice!
greet("Bob")    # Output: Hello, Bob!
```

Explanation:

1. Defining the Function:
    - Use the def keyword to define a function named greet that accepts one parameter, name.
    - The docstring describes what the function does.
    - The function body contains a print statement that prints a greeting message.

2. Calling the Function:

- Call the greet function with the argument "Alice". The function prints "Hello, Alice!".
- Call the greet function with the argument "Bob". The function prints "Hello, Bob!".

Function Arguments

1. Functions can accept different types of arguments, including:
2. Positional Arguments: Arguments that are passed to the function in the correct positional order.
3. Keyword Arguments: Arguments that are passed to the function with their parameter name.
4. Default Arguments: Arguments that assume a default value if a value is not provided in the function call.
5. Variable-length Arguments: Functions that can accept a variable number of arguments using *args (for non-keyword arguments) and **kwargs (for keyword arguments).

Example of Different Types of Arguments:

```
# Function with different types of arguments
def example_function(positional, keyword="default", *args, **kwargs):
    print(f"Positional: {positional}")
    print(f"Keyword: {keyword}")
    print(f"Variable-length args: {args}")
    print(f"Variable-length kwargs: {kwargs}")

# Calling the function
example_function(1, "custom", 2, 3, key1="value1", key2="value2")
```

Explanation:

- Define a function named example_function with different types of arguments.
- Print statements to display the values of the arguments.
- Call the function with various arguments and print the results.

6.2. Parameters and Arguments

Parameters and arguments are essential components of functions in Python. They allow you to pass data to a function so that the function can process it and return a result. Understanding how to use parameters and arguments effectively can make your code more flexible and reusable.

Parameters

Parameters are variables that are defined in the function signature (the part of the function definition that includes the function name and its parameter list). Parameters act as placeholders for the values that will be passed to the function when it is called.

Example of Parameters:

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

In this example, name is a parameter.

Arguments

Arguments are the actual values that are passed to the function when it is called. These values are assigned to the corresponding parameters in the function definition.

Example of Arguments:

```
greet("Alice")  # "Alice" is the argument
```

In this example, "Alice" is an argument.

Types of Parameters and Arguments

1. **Positional Arguments**
2. **Keyword Arguments**
3. **Default Arguments**

4. **Variable-length Arguments**

1. Positional Arguments:

Positional arguments are the most common type of arguments. They are passed to the function in the same order as the parameters are defined.

Example 1:

```
def multiply(a, b):
    return a * b

# Calling the function
result = multiply(4, 5)
print(result)  # Output: 20
```

Explanation:

- The multiply function takes two positional arguments ‘a’ and ‘b’.
- The values ‘4’ and ‘5’ are passed to the function as positional arguments.
- The function returns the product of ‘a’ and ‘b’.

Example 2:

```
def concatenate(str1, str2):
    return str1 + str2

# Calling the function
result = concatenate("Hello, ", "World!")
print(result)  # Output: Hello, World!
```

Explanation:

- The concatenate function takes two positional arguments ‘str1’ and ‘str2’.
- The values "Hello, " and "World!" are passed to the function as positional arguments.
- The function returns the concatenated string.

Example 3:

```
def calculate_area(length, width):
    return length * width

# Calling the function
area = calculate_area(10, 5)
print(f"The area of the rectangle is: {area} square units")  # Output: The
area of the rectangle is: 50 square units
```

Explanation:

- The calculate_area function takes two positional arguments length and width.
- The values 10 and 5 are passed to the function as positional arguments.
- The function returns the area of the rectangle.

2. Keyword Arguments:

Keyword arguments are passed to the function with their parameter names. This allows you to pass arguments in any order.

Example1:

```
def introduce(name, age):
    print(f"Name: {name}, Age: {age}")

# Calling the function with keyword arguments
introduce(age=30, name="Alice")  # Output: Name: Alice, Age: 30
```

Explanation:

- The introduce function takes two parameters 'name' and 'age'.
- The keyword arguments 'age=30' and 'name="Alice"' are passed to the function.
- The function prints the 'name' and 'age'.

Example 2:

```
def order(item, quantity):
    print(f"Order placed for {quantity} units of {item}.")

# Calling the function with keyword arguments
order(quantity=3, item="apples")  # Output: Order placed for 3 units of
apples.
```

Explanation:

- The order function takes two parameters 'item' and 'quantity'.
- The keyword arguments 'quantity=3' and 'item="apples"' are passed to the function.
- The function prints the order details.

Example 3:

```
def book_ticket(movie, time, seat):
    print(f"Ticket booked for {movie} at {time}. Seat: {seat}")

# Calling the function with keyword arguments
book_ticket(time="7:00 PM", movie="Inception", seat="A12")
# Output: Ticket booked for Inception at 7:00 PM. Seat: A12
```

Explanation:

- The book_ticket function takes three parameters 'movie',' time', and 'seat'.
- The keyword arguments 'time="7:00 PM"', 'movie="Inception"', and 'seat="A12"' are passed to the function.
- The function prints the ticket booking details.

3. Default Arguments

Default arguments allow you to assign default values to parameters. If no argument is passed for a parameter with a default value, the default value is used.

Example:

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

# Calling the function without an argument
greet()  # Output: Hello, Guest!

# Calling the function with an argument
greet("Alice")  # Output: Hello, Alice!
```

Explanation:

- The greet function has a default parameter name with the default value "Guest".
- When the function is called without an argument, the default value is used.
- When the function is called with an argument, the provided value is used.

Example 2:

```
def display_info(name, age=18):
    print(f"Name: {name}, Age: {age}")

# Calling the function with one argument
display_info("Bob")  # Output: Name: Bob, Age: 18

# Calling the function with two arguments
display_info("Alice", 25)  # Output: Name: Alice, Age: 25
```

Explanation:

- The display_info function has a parameter age with the default value 18.
- When the function is called with one argument, the default value of age is used.
- When the function is called with two arguments, the provided values are used.

Example 3:

```
def calculate_discount(price, discount=0.10):
```

```
    final_price = price - (price * discount)
    return final_price

# Calling the function with one argument
print(calculate_discount(100))  # Output: 90.0

# Calling the function with two arguments
print(calculate_discount(100, 0.20))  # Output: 80.0
```

Explanation:

- The calculate_discount function has a parameter discount with the default value 0.10 (10%).
- When the function is called with one argument, the default discount is applied.
- When the function is called with two arguments, the provided discount is applied.

4. Variable-length Arguments

Variable-length arguments allow you to pass a variable number of arguments to a function. There are two types:

1. ***args: For non-keyword variable-length arguments.**
2. ****kwargs: For keyword variable-length arguments.**

Using *args

*args allows you to pass a variable number of non-keyword arguments to a function. Inside the function, args is a tuple of the arguments passed.

Example 1:

```
def sum_numbers(*args):
    return sum(args)

# Calling the function with different numbers of arguments
```

```
print(sum_numbers(1, 2, 3))  # Output: 6
print(sum_numbers(4, 5))     # Output: 9
```

Explanation:

- The sum_numbers function uses *args to accept a variable number of non-keyword arguments.
- The arguments are passed as a tuple to the function, and the sum of the numbers is returned.

Example 2:

```
def favorite_foods(*foods):
    for food in foods:
        print(food)

# Calling the function with different numbers of arguments
favorite_foods("Pizza", "Burger", "Ice Cream")
# Output:
# Pizza
# Burger
# Ice Cream
```

Explanation:

- The favorite_foods function uses *foods to accept a variable number of non-keyword arguments.
- The arguments are passed as a tuple to the function, and each food item is printed.

Example 3:

```
def record_scores(*scores):
    for i, score in enumerate(scores, start=1):
        print(f"Score {i}: {score}")

# Calling the function with different numbers of arguments
record_scores(89, 92, 78, 94)
```

```
# Output:
# Score 1: 89
# Score 2: 92
# Score 3: 78
# Score 4: 94
```

Explanation:

- The record_scores function uses *scores to accept a variable number of non-keyword arguments.
- The arguments are passed as a tuple to the function, and each score is printed with its corresponding position.

2. Using **kwargs

**kwargs allows you to pass a variable number of keyword arguments to a function. Inside the function, kwargs is a dictionary of the arguments passed.

Example 1:

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Calling the function with different keyword arguments
print_info(name="Alice", age=30, city="New York")
# Output:
# name: Alice
# age: 30
# city: New York
```

Explanation:

- The print_info function uses **kwargs to accept a variable number of keyword arguments.
- The arguments are passed as a dictionary to the function, and each key-value pair is printed.

Example 2:

```
def display_config(**config):
    for key, value in config.items():
        print(f"{key}: {value}")

# Calling the function with different keyword arguments
display_config(resolution="1080p", brightness=70, volume=50)
# Output:
# resolution: 1080p
# brightness: 70
# volume: 50
```

Explanation:

- The display_config function uses **config to accept a variable number of keyword arguments.
- The arguments are passed as a dictionary to the function, and each key-value pair is printed.

Example 3:

```
def register_user(**user_details):
    print("User Registration Details:")
    for key, value in user_details.items():
        print(f"{key}: {value}")

# Calling the function with different keyword arguments
register_user(username="john_doe", email="john@example.com", age=28,
country="USA")
# Output:
# User Registration Details:
# username: john_doe
# email: john@example.com
# age: 28
# country: USA
```

Explanation:

- The register_user function uses **user_details to accept a variable number of keyword arguments.
- The arguments are passed as a dictionary to the function, and each key-value pair is printed.

Real World Example for Parameter and Arguments:

Example 1: Order Summary

```
def order_summary(order_id, *items, **details):
    print(f"Order ID: {order_id}")
    print("Items:")
    for item in items:
        print(f" - {item}")
    print("Details:")
    for key, value in details.items():
        print(f" {key}: {value}")

# Calling the function
order_summary(101, "Pizza", "Soda", "Fries", customer="Alice", table=5,
time="7:00 PM")

# Output:
# Order ID: 101
# Items:
#  - Pizza
#  - Soda
#  - Fries
# Details:
#  customer: Alice
#  table: 5
#  time: 7:00 PM
```

Explanation:

- The order_summary function accepts a positional argument order_id, variable-length non-keyword arguments *items, and variable-length keyword arguments **details.
- The order ID, items, and additional details are printed.

Example 2: Employee Profile

```
def employee_profile(employee_id, name, position="Staff", *skills,
**additional_info):
    print(f"Employee ID: {employee_id}")
    print(f"Name: {name}")
    print(f"Position: {position}")
    print("Skills:")
    for skill in skills:
        print(f" - {skill}")
    print("Additional Info:")
    for key, value in additional_info.items():
        print(f" {key}: {value}")

# Calling the function
employee_profile(123, "Bob", "Manager", "Leadership", "Time Management",
department="Sales", tenure="5 years")
# Output:
# Employee ID: 123
# Name: Bob
# Position: Manager
# Skills:
#  - Leadership
#  - Time Management
# Additional Info:
#  department: Sales
#  tenure: 5 years
```

Explanation:

- The employee_profile function accepts positional arguments employee_id and name, a default argument position, variable-length non-keyword arguments *skills, and variable-length keyword arguments **additional_info.

- The employee's profile details, skills, and additional information are printed.

6.3. Return Statements

The return statement is used in functions to send back a result or value to the caller. When a return statement is executed, the function terminates, and the specified value is returned to the caller. If no return statement is used, the function returns None by default.

Example 1:

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

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

Explanation:

- The add function takes two parameters, ‘a’ and ‘b’.
- The return statement returns the sum of ‘a’ and ‘b’.
- The result is stored in the variable result and printed.

Example 2:

```
def is_even(number):
    if number % 2 == 0:
        return True
    else:
        return False

# Calling the function
print(is_even(4))  # Output: True
print(is_even(7))  # Output: False
```

Explanation:

- The is_even function checks if a number is even.
- If the number is even, it returns True; otherwise, it returns False.
- The function's return value is printed.

Multiple Return Statements

A function can have multiple return statements. The function exits as soon as one of the return statements is executed.

Example:

```
def classify_number(number):
    if number > 0:
        return "Positive"
    elif number < 0:
        return "Negative"
    else:
        return "Zero"

# Calling the function
print(classify_number(10))  # Output: Positive
print(classify_number(-5))  # Output: Negative
print(classify_number(0))   # Output: Zero
```

Explanation:

- The classify_number function classifies a number as positive, negative, or zero.
- It has multiple return statements, one for each condition.
- The function exits as soon as one of the return statements is executed.
- Returning Multiple Values
- A function can return multiple values using a tuple.

Example:

```
def get_person_info():
    name = "Alice"
    age = 30
    city = "New York"
    return name, age, city
```

```
# Calling the function
info = get_person_info()
print(info)  # Output: ('Alice', 30, 'New York')

# Unpacking the returned tuple
name, age, city = get_person_info()
print(name)  # Output: Alice
print(age)   # Output: 30
print(city)  # Output: New York
```

Explanation:

- The 'get_person_info' function returns multiple values as a tuple.
- The returned tuple is stored in the variable 'info' and printed.
- The returned tuple can also be unpacked into separate variables.

Real-World Examples

Example 1: Calculating Grades

```
def calculate_grade(marks):
    if marks >= 90:
        return 'A'
    elif marks >= 80:
        return 'B'
    elif marks >= 70:
        return 'C'
    elif marks >= 60:
        return 'D'
    else:
        return 'F'

# Calling the function
print(calculate_grade(85))  # Output: B
print(calculate_grade(72))  # Output: C
print(calculate_grade(58))  # Output: F
```

Explanation:

- The calculate_grade function returns a grade based on the marks.
- It uses multiple return statements to return different grades.
- The function's return value is printed.

Example 2: Employee Performance Review

```
def performance_review(employee_name, sales):
    if sales >= 100000:
        return employee_name, "Outstanding"
    elif sales >= 75000:
        return employee_name, "Excellent"
    elif sales >= 50000:
        return employee_name, "Good"
    elif sales >= 25000:
        return employee_name, "Average"
    else:
        return employee_name, "Needs Improvement"

# Calling the function
name, performance = performance_review("John Doe", 85000)
print(f"{name}: {performance}")  # Output: John Doe: Excellent

name, performance = performance_review("Jane Smith", 23000)
print(f"{name}: {performance}")  # Output: Jane Smith: Needs Improvement
```

Explanation:

- The performance_review function returns the employee's name and performance level based on sales.
- It uses multiple return statements to return different performance levels.
- The function's return value is unpacked and printed.

6.4. Scope of Variables

The scope of a variable determines the part of the program where the variable can be accessed or modified. In Python, variables can have local or global scope.

Local Scope

A variable defined inside a function has a local scope and is accessible only within that function.

Example 1:

```
def greet():
    message = "Hello, World!"
    print(message)

greet()  # Output: Hello, World!
print(message)  # Error: NameError: name 'message' is not defined
```

Explanation:

- The variable message is defined inside the greet function, giving it local scope.
- The message variable is accessible only within the greet function.
- Trying to access message outside the function results in a NameError.

Example 2:

```
def add_numbers(a, b):
    result = a + b
    return result

print(add_numbers(5, 3))  # Output: 8
print(result)  # Error: NameError: name 'result' is not defined
```

Explanation:

- The variable result is defined inside the add_numbers function, giving it local scope.
- The result variable is accessible only within the add_numbers function.

- Trying to access result outside the function results in a NameError.

Global Scope

A variable defined outside any function has a global scope and is accessible throughout the program.

Example 1:

```
global_message = "Hello, World!"

def greet():
    print(global_message)

greet()  # Output: Hello, World!
print(global_message)  # Output: Hello, World!
```

Explanation:

- The variable global_message is defined outside the greet function, giving it global scope.
- The global_message variable is accessible both inside and outside the greet function.

Example 2:

```
result = 0

def add_numbers(a, b):
    global result
    result = a + b

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

Explanation:

- The variable result is defined outside the add_numbers function, giving it global scope.
- The global keyword is used inside the add_numbers function to modify the global variable result.
- The modified value of result is accessible outside the function.

Non-local Scope

The nonlocal keyword is used to work with variables inside nested functions, where the variable should not be in the local or global scope.

Example 1:

```
def outer_function():
    outer_var = "I am outside!"

    def inner_function():
        nonlocal outer_var
        outer_var = "I am inside!"

    inner_function()
    print(outer_var)

outer_function()  # Output: I am inside!
```

Explanation:

- The variable outer_var is defined in the outer_function, making it local to that function.
- The inner_function modifies outer_var using the nonlocal keyword.
- The modified value of outer_var is printed in the outer_function.

Real-World Examples

Example 1: Configuration Settings

```
config = {
    "setting1": "value1",
    "setting2": "value2",
    "setting3": "value3"
}

def update_config(key, value):
    global config
    config[key] = value

update_config("setting2", "new_value2")
print(config)
# Output: {'setting1': 'value1', 'setting2': 'new_value2', 'setting3':
'value3'}
```

Explanation:

- The config dictionary is defined globally.
- The update_config function uses the global keyword to modify the global config dictionary.
- The updated configuration settings are printed.

Example 2: Counter Function

```
def create_counter():
    count = 0

    def increment():
        nonlocal count
        count += 1
        return count

    return increment

counter = create_counter()
print(counter())  # Output: 1
print(counter())  # Output: 2
```

```
print(counter())  # Output: 3
```

Explanation:

- The count variable is defined in the create_counter function, making it local to that function.
- The increment function modifies count using the nonlocal keyword.
- The create_counter function returns the increment function, which is then called to increment the counter.

6.5. Lambda function

- A Lambda function in Python is a small, anonymous (unnamed) function defined with the lambda keyword.
- It can take any number of arguments, but can only have one expression.
- Lambda functions are used for creating small, quick functions that are not needed elsewhere in your code.
- They are typically used for operations that are simple and short enough to be expressed in a single line.
- Conciseness: Lambda functions allow you to write functions in a more concise manner compared to traditional def functions.
- Readability: They can make your code more readable when the function logic is straightforward and does not require multiple lines

Key Characteristics:

- Anonymous: Lambda functions are anonymous because they don't require a name like regular functions defined with def.
- Single Expression: They can only contain a single expression, the result of which is returned.
- Conciseness: They are concise and often used for functions that are simple and small in scope.

Syntax:

```
lambda arguments: expression
```

- lambda: Keyword used to define a lambda function.
- arguments: Comma-separated list of parameters (similar to a regular function).
- expression: Single expression whose result is returned by the function.

Examples:

1. Adding Two Numbers:

```
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8
```

Explanation:

- Here, lambda x, y: x + y defines a lambda function that takes two arguments x and y, and returns their sum.
- add(5, 3) calls the lambda function with arguments 5 and 3, returning 8.

2. Checking if a Number is Even

```
is_even = lambda x: x % 2 == 0
print(is_even(6))  # Output: True
print(is_even(7))  # Output: False
```

Explanation:

- lambda x: x % 2 == 0 defines a lambda function that checks if x is even by evaluating x % 2 == 0.
- is_even(6) returns True because 6 is even, and is_even(7) returns False because 7 is odd.

3. Sorting a List of Tuples by the Second Element

```
points = [(1, 2), (3, 1), (2, 3)]
points_sorted = sorted(points, key=lambda x: x[1])
print(points_sorted)  # Output: [(3, 1), (1, 2), (2, 3)]
```

Explanation:

- lambda x: x[1] is used as the key function for sorting points based on the second element of each tuple.
- sorted(points, key=lambda x: x[1]) sorts points based on the second element of each tuple, resulting in [(3, 1), (1, 2), (2, 3)].

Problem statement:

Create a lambda function to compute the area of a rectangle given its length and width. Then, use the lambda function to calculate the area of a rectangle with length 5 units and width 3 units.

Solution:

```
area_rectangle = lambda length, width: length * width
area = area_rectangle(5, 3)
print("Area of rectangle:", area)  # Output: Area of rectangle: 15
```

Explanation:

- area_rectangle = lambda length, width: length * width defines a lambda function to calculate the area of a rectangle given its length and width.
- area_rectangle(length, width) calculates the area of a rectangle with length = 5 and width = 3, resulting in an area of 15.

6.6. Map function

- map() function in Python applies a given function to all items in an iterable (like lists, tuples) and returns an iterator that yields the results.
- Purpose: map() is used to apply a function to all elements of an iterable without using explicit loops.
- Efficiency: It provides a concise way to transform data, especially useful when working with large datasets or functional programming paradigms.

Key Characteristics:

- Returns an Iterator: It doesn't compute the values immediately; instead, it returns an iterator that generates values when needed.
- One-to-One Mapping: Applies the function to each element of the iterable in sequence.
- Supports Multiple Iterables: map() can take multiple iterables; the function must then have as many arguments as there are iterables.

Syntax:

```
map(function, iterable1, iterable2, ...)
```

- function: A function to apply to each element of the iterable(s).
- iterable1, iterable2, ...: One or more iterables (like lists, tuples) whose elements will be passed as arguments to function.

We use lambda sometimes inside map(), because, lambda functions inside map() are used for concise and inline transformations of iterable elements, enhancing code readability and maintainability by avoiding the need for separate function definitions for simple operations.

Syntax of using lambda() inside map():

```
map(lambda arguments: expression, iterable)
```

Examples:

1. Square Numbers using map()

```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16, 25]
```

Explanation:

- map(lambda x: x ** 2, numbers) applies the lambda function to each element of numbers, squaring each element.
- list(squared) converts the iterator squared into a list, resulting in [1, 4, 9, 16, 25].

2. Concatenate First and Last Names

```
first_names = ['Alice', 'Bob', 'Charlie']
last_names = ['Smith', 'Johnson', 'Brown']
full_names = map(lambda x, y: x + ' ' + y, first_names, last_names)
print(list(full_names))
# Output: ['Alice Smith', 'Bob Johnson', 'Charlie Brown']
```

Explanation:

- map(lambda x, y: x + ' ' + y, first_names, last_names) concatenates corresponding elements from first_names and last_names.
- list(full_names) converts the iterator full_names into a list, resulting in ['Alice Smith', 'Bob Johnson', 'Charlie Brown'].

4. Mapping with a Named Function

```
def square(x):
    return x ** 2

numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(square, numbers))
print(squared_numbers)
# Output: [1, 4, 9, 16, 25]
```

Explanation:

- map(square, numbers) applies the square function to calculate the square of each number in numbers.
- list(map(square, numbers)) converts the iterator to a list, resulting in [1, 4, 9, 16, 25].

Problem Statement:

Convert a list of temperatures in Celsius to Fahrenheit using the map function.

Solution:

```
# List of temperatures in Celsius
celsius_temps = [0, 10, 20, 30, 40]

# Convert Celsius to Fahrenheit using map and lambda function
fahrenheit_temps = list(map(lambda c: (9/5) * c + 32, celsius_temps))

# Print the converted temperatures
print("Celsius Temperatures:", celsius_temps)
print("Fahrenheit Temperatures:", fahrenheit_temps)

#Output:
Celsius Temperatures: [0, 10, 20, 30, 40]
Fahrenheit Temperatures: [32.0, 50.0, 68.0, 86.0, 104.0]
```

Explanation:

- map(lambda c: (9/5) * c + 32, celsius_temps) applies the lambda function to each Celsius temperature in celsius_temps, converting it to Fahrenheit.
- list(map(lambda c: (9/5) * c + 32, celsius_temps)) converts the iterator returned by map into a list of Fahrenheit temperatures.

7. Modules and Packages

7.1. Importing Modules:

modules are files containing Python code, which can define functions, classes, and variables that can be utilized in other Python programs. Importing modules allows you to reuse code and organize your Python projects into manageable components.

1. Basic Module Import

To use code from a module in your Python program, you need to import it using the import statement followed by the module name.

Example: Using the math module

```
import math

print(math.sqrt(25))  # Output: 5.0 (square root of 25)
print(math.pi)        # Output: 3.141592653589793 (value of pi)
```

2. Importing Specific Items

You can import specific functions, classes, or variables from a module instead of importing the entire module. This approach can save memory and reduce potential name conflicts.

Example: Importing specific items from the math module

```
from math import sqrt, pi

print(sqrt(25))  # Output: 5.0 (square root of 25)
print(pi)        # Output: 3.141592653589793 (value of pi)
```

3. Alias for Module or Items

You can provide an alias for a module or its items during import, which can make your code more readable or help resolve naming conflicts.

Example: Using an alias for a module

```
import math as m

print(m.sqrt(25))  # Output: 5.0 (square root of 25)
print(m.pi)        # Output: 3.141592653589793 (value of pi)
```

4. Importing All Items

You can import all items from a module into the current namespace using the from module_name import * syntax. However, this is generally discouraged because it can lead to name conflicts and make it unclear where functions or constants come from.

Example: Importing all items from a module

```
from math import *

print(sqrt(25))  # Output: 5.0 (square root of 25)
print(pi)        # Output: 3.141592653589793 (value of pi)
```

Real-World Examples

Example 1: Using the datetime module

```
from datetime import datetime

current_time = datetime.now()
print(current_time)  # Output: 2024-07-05 12:00:00.000000
```

Example 2: Working with the random module

```
import random

random_number = random.randint(1, 100)
print(random_number)  # Output: a random number between 1 and 100
```

7.2. Creating your own modules

Creating your own modules in Python allows you to organize your code into separate files, making it more manageable, reusable, and easier to understand. A module in Python is simply a file with a .py extension that contains Python code, such as functions, classes, and variables.

Steps to Create and Use Your Own Module

1. Create a Python file:

- Write the code you want to include in your module and save it with a .py extension.

2. Import the module in another Python file:

- Use the import statement to include your module in another Python file and use its functions, classes, or variables.

Example: Creating and Using a Custom Module

Step 1: Create a module file (mymodule.py):

```
# mymodule.py

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

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

def subtract(a, b):
    return a - b

pi = 3.14159
```

Explanation:

- The mymodule.py file defines three functions (greet, add, and subtract) and one variable (pi).

Step 2: Use the module in another file (main.py):

```
# main.py

import mymodule

print(mymodule.greet("Alice"))  # Output: Hello, Alice!
print(mymodule.add(5, 3))       # Output: 8
print(mymodule.subtract(10, 4)) # Output: 6
print(mymodule.pi)              # Output: 3.14159
```

Explanation:

- The main.py file imports the mymodule module and uses its functions and variable.
- You access the functions and variable from mymodule using dot notation (mymodule.function_name or mymodule.variable_name).

Using Specific Items from a Module

You can import specific functions, classes, or variables from your module using the from module_name import item_name syntax.

Example:

```
# main.py

from mymodule import greet, pi

print(greet("Bob"))  # Output: Hello, Bob!
print(pi)            # Output: 3.14159
```

Explanation:

- The from mymodule import greet, pi statement imports only the greet function and pi variable from the mymodule module.

- You can use greet and pi directly without prefixing mymodule..

Aliasing Your Module or Items

- You can provide an alias for your module or its items to make your code more readable or to resolve naming conflicts.

Example: Using an alias for a module

```
# main.py

import mymodule as mm

print(mm.greet("Charlie"))  # Output: Hello, Charlie!
print(mm.pi)                # Output: 3.14159
```

Explanation:

- The import mymodule as mm statement imports the mymodule module with an alias mm.
- Functions and variables from the mymodule module are accessed using mm. instead of mymodule..

Example: Using an alias for specific items

```
# main.py

from mymodule import add as add_numbers, pi as circle_constant

print(add_numbers(2, 3))     # Output: 5
print(circle_constant)       # Output: 3.14159
```

Explanation:

- The from mymodule import add as add_numbers, pi as circle_constant statement imports the add function and pi variable with aliases.
- You can use add_numbers and circle_constant directly in your code.

Real-World Example

Example: Utility Module

Create a utility module with common functions and use it in another program.

Step 1: Create the utility module (utility.py):

```
# utility.py

def read_file(file_path):
    with open(file_path, 'r') as file:
        return file.read()

def write_file(file_path, content):
    with open(file_path, 'w') as file:
        file.write(content)

def calculate_average(numbers):
    return sum(numbers) / len(numbers) if numbers else 0
```

Explanation:

- The utility.py file defines three functions: read_file, write_file, and calculate_average.

Step 2: Use the utility module in another program (main.py):

```
# main.py

import utility

content = utility.read_file('example.txt')
print("File Content:", content)

utility.write_file('output.txt', 'Hello, World!')

numbers = [10, 20, 30, 40]
average = utility.calculate_average(numbers)
print("Average:", average)  # Output: Average: 25.0
```

Explanation:

- The main.py file imports the utility module and uses its functions to read from a file, write to a file, and calculate the average of a list of numbers.

7.3. Exploring Python's Standard Library

Python's standard library is a collection of modules and packages included with Python, providing various functionalities to help you accomplish common tasks. The standard library covers areas such as file I/O, system calls, data manipulation, internet protocols, and more, allowing you to perform many tasks without needing third-party packages.

Commonly Used Standard Library Modules

- os: Interacting with the operating system.
- sys: Accessing system-specific parameters and functions.
- math: Performing mathematical operations.
- datetime: Manipulating dates and times.
- random: Generating random numbers.
- json: Working with JSON data.
- re: Handling regular expressions.
- collections: Providing alternative container types.
- itertools: Creating iterators for efficient looping.
- functools: Higher-order functions for functional programming.
- threading: Creating and managing threads.
- multiprocessing: Running parallel processes.
- subprocess: Running subprocesses.
- http.client: Handling HTTP requests.
- unittest: Writing and running tests.

Detailed Examples

1. 'os' Module

The os module provides a way to interact with the operating system, allowing you to perform tasks such as reading and writing files, navigating directories, and managing environment variables.

Example: Using the os module

```
import os

# Get the current working directory
current_directory = os.getcwd()
print("Current Directory:", current_directory)

# List all files and directories in the current directory
items = os.listdir(current_directory)
print("Items in Directory:", items)

# Create a new directory
os.mkdir("new_directory")

# Remove a directory
os.rmdir("new_directory")

#Output:
Current Directory: /home/user
Items in Directory: ['file1.txt', 'file2.txt', 'new_directory']
```

- os.getcwd() returns the current working directory.
- os.listdir(path) lists all files and directories in the specified path.
- os.mkdir(path) creates a new directory.
- os.rmdir(path) removes an empty directory.

2. 'sys' Module

The sys module provides access to some variables used or maintained by the Python interpreter and functions that interact strongly with the interpreter.

Example: Using the sys module

```
import sys

# Get the list of command-line arguments
```

```
arguments = sys.argv
print("Command-line Arguments:", arguments)

# Get the Python version
python_version = sys.version
print("Python Version:", python_version)

# Exit the script
sys.exit("Exiting script")

#Output:
Command-line Arguments: ['script.py', 'arg1', 'arg2']
Python Version: 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0]
```

- sys.argv is a list of command-line arguments passed to the script.
- sys.version returns the Python version.
- sys.exit(message) exits the script and optionally prints a message.

3. 'math' Module

The math module provides mathematical functions and constants.

Example: Using the math module

```
import math

# Calculate the square root of a number
sqrt_value = math.sqrt(16)
print("Square Root:", sqrt_value)

# Calculate the sine of an angle in radians
sine_value = math.sin(math.pi / 2)
print("Sine:", sine_value)

# Get the value of pi
pi_value = math.pi
```

```
print("Pi:", pi_value)

#output:
Square Root: 4.0
Sine: 1.0
Pi: 3.141592653589793
```

- math.sqrt(x) returns the square root of x.
- math.sin(x) returns the sine of x radians.
- math.pi returns the value of pi.

4. 'datetime' Module

The datetime module supplies classes for manipulating dates and times.

Example: Using the datetime module

```
from datetime import datetime, timedelta

# Get the current date and time
now = datetime.now()
print("Current Date and Time:", now)

# Format the date and time
formatted_now = now.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted Date and Time:", formatted_now)

# Calculate a future date
future_date = now + timedelta(days=10)
print("Future Date:", future_date)

#output:
Current Date and Time: 2024-07-05 15:30:00
Formatted Date and Time: 2024-07-05 15:30:00
Future Date: 2024-07-15 15:30:00
```

- datetime.now() returns the current date and time.
- datetime.strftime(format) formats the date and time.
- timedelta is used to perform date arithmetic.

5. 'json' Module

The json module provides functions for parsing JSON strings and converting data to JSON format.

Example: Using the json module

```
import json

# Convert a Python dictionary to a JSON string
data = {"name": "John", "age": 30, "city": "New York"}
json_string = json.dumps(data)
print("JSON String:", json_string)

# Parse a JSON string to a Python dictionary
parsed_data = json.loads(json_string)
print("Parsed Data:", parsed_data)

#Output:
JSON String: {"name": "John", "age": 30, "city": "New York"}
Parsed Data: {'name': 'John', 'age': 30, 'city': 'New York'}
```

- json.dumps(obj) converts a Python object to a JSON string.
- json.loads(json_string) parses a JSON string to a Python object.

6. 're' Module

The re module provides support for regular expressions.

Example: Using the re module

```
import re

# Define a pattern and a string
pattern = r"\b\d{3}-\d{2}-\d{4}\b"
text = "My phone number is 123-45-6789."

# Search for the pattern in the string
match = re.search(pattern, text)
if match:
    print("Match Found:", match.group())
else:
    print("No Match Found")

#Output:
Match Found: 123-45-6789
```

- re.search(pattern, string) searches for the pattern in the string and returns a match object if found.
- match.group() returns the matched string.

We have more modules:

- csv: Working with CSV (Comma Separated Values) files.
- pickle: Serializing and deserializing Python objects.
- sqlite3: Interacting with SQLite databases.
- socket: Networking support for network communication.
- xml.etree.ElementTree: Manipulating XML data.
- argparse: Parsing command-line arguments.
- logging: Flexible logging framework.
- timeit: Timing the execution of small code snippets.
- gzip: Reading and writing gzip-compressed files.
- zipfile: Working with ZIP archives.
- email: Handling email messages and MIME attachments.

- hashlib: Generating secure hashes and message digests.
- html.parser: Parsing HTML and XML documents.

Each of these modules provides specialized functionality that can be extremely useful in different scenarios, from data processing and file handling to networking and security. And we have many more modules, these are some modules we use mostly.

8. File Handling

File handling in Python refers to the various operations that can be performed on files, such as reading from and writing to files, manipulating file metadata, and managing file objects. Python provides built-in functions and methods within the standard library that simplify file operations, making it easy to work with files stored on disk or accessed over a network.

Operations in File Handling:

1. **Opening Files:** Files are typically opened using the open() function, which returns a file object representing the file on disk.
2. **Reading from Files:** Once a file is opened, its contents can be read using methods like read(), readline(), or readlines().
3. **Writing to Files:** Files can be written to using methods like write() or writelines().
4. **Closing Files:** It's important to close files after reading from or writing to them to release system resources. This is typically done using the close() method on the file object.
5. **Managing Files with Context Managers:** Python's with statement allows for safer and more concise management of file objects by automatically closing the file when the block of code is exited.
6. **File Modes:** Files can be opened in different modes ('r' for reading, 'w' for writing, 'a' for appending, 'b' for binary mode, etc.), depending on the operation you want to perform.
7. **File Navigation:** Python provides methods to navigate within a file, such as seek() and tell().
8. **Exception Handling:** File operations can raise exceptions, so it's important to handle errors using try, except, finally blocks.

8.1. Reading from Files

Reading from files in Python involves opening a file in read mode ('r') and then accessing its contents. There are several methods available to read data from a file, depending on how you want to handle the file's contents.

Methods for Reading Files:

1. **read():** Reads the entire file content as a single string.
2. **readline():** Reads a single line from the file.
3. **readlines():** Reads all lines in the file into a list where each line is a list item.

Example: Using read()

```
# Open a file in read mode
file_path = 'sample.txt'
with open(file_path, 'r') as file:
    file_content = file.read()
    print("File Content:")
    print(file_content)

#Output:
File Content:
This is line 1.
This is line 2.
This is line 3.
```

- open(file_path, 'r') opens the file sample.txt in read mode.
- file.read() reads the entire content of the file and stores it in file_content.
- The with statement ensures that the file is properly closed after its suite finishes, even if an exception is raised.

Example: Using readline()

```
# Open a file in read mode
file_path = 'sample.txt'
with open(file_path, 'r') as file:
    line = file.readline()
    while line:
        print("Line:", line.strip())
        line = file.readline()

#Output:
Line: This is line 1.
Line: This is line 2.
Line: This is line 3.
```

- file.readline() reads a single line from the file each time it's called.
- The while line: loop continues until file.readline() returns an empty string, indicating the end of the file.
- line.strip() removes any leading and trailing whitespace characters from the line before printing.

Example: Using readlines()

```
# Open a file in read mode
file_path = 'sample.txt'
with open(file_path, 'r') as file:
    lines = file.readlines()
    for line_num, line in enumerate(lines):
        print(f"Line {line_num + 1}: {line.strip()}")

#Output:
Line 1: This is line 1.
Line 2: This is line 2.
Line 3: This is line 3.
```

- file.readlines() reads all lines from the file and returns them as a list of strings, where each string represents a line.

- enumerate(lines) allows iterating over the list of lines while also getting their index (line_num).
- line.strip() removes leading and trailing whitespace characters from each line before printing.

8.2. Writing to Files

Writing to files in Python involves opening a file in write mode ('w', which truncates the file if it exists) or append mode ('a', which appends to the end of the file if it exists), and then writing data to it. This section covers various methods to write data to files and examples illustrating their usage.

Methods for Writing Files:

1. **'write()':** Writes a string to the file.
2. **'writelines()':** Writes a list of strings to the file.

Example: Using 'write()'

```
# Open a file in write mode
file_path = 'output.txt'
with open(file_path, 'w') as file:
    file.write("This is line 1.\n")
    file.write("This is line 2.\n")
    file.write("This is line 3.\n")
```

- 'open(file_path, 'w')' opens the file output.txt in write mode.
- 'file.write("...")' writes the specified string to the file.

Each write() statement adds a line to the file due to the newline character \n at the end of each string.

Example: Using writelines()

```
# Open a file in write mode
file_path = 'output.txt'
lines_to_write = ["First line.\n", "Second line.\n", "Third line.\n"]
with open(file_path, 'w') as file:
    file.writelines(lines_to_write)
```

- ‘file.writelines(lines_to_write)’ writes each string in the list lines_to_write to the file without adding any additional characters between them.
- This method is useful for writing multiple lines at once.

8.3. Using Context Managers (with Statement)

Python's 'with' statement provides a convenient way to manage resources, such as files, by ensuring they are properly opened and closed. It simplifies exception handling and ensures that cleanup actions are performed. This section demonstrates how to use the with statement for file handling.

Example: Using 'with' Statement for File Handling

```
# Example 1: Reading from a File
file_path = 'sample.txt'
with open(file_path, 'r') as file:
    file_content = file.read()
    print("File Content:")
    print(file_content)
```

- The 'with open(file_path, 'r') as file': statement opens the file 'sample.txt' in read mode ('r').
- Inside the 'with' block, 'file.read()' reads the entire content of the file and stores it in 'file_content'.
- The 'with' statement automatically closes the file ('file.close()') once the block of code is executed, even if an exception occurs.

Example: Writing to a File with 'with' Statement

```
# Example 2: Writing to a File
file_path = 'output.txt'
lines_to_write = ["First line.\n", "Second line.\n", "Third line.\n"]
with open(file_path, 'w') as file:
    file.writelines(lines_to_write)
    print(f"{len(lines_to_write)} lines written to {file_path}")
```

- The 'with open(file_path, 'w') as file': statement opens the file 'output.txt' in write mode ('w').
- 'file.writelines(lines_to_write)' writes the list 'lines_to_write' to the file.

- The ‘print’ statement confirms how many lines were written to the file.

9. Object-Oriented Programming (OOP)

9.1. Introduction to OOP

Object-Oriented Programming (OOP) is a paradigm in which programs are organized as cooperative collections of objects, each representing an instance of a class. This paradigm encourages the decomposition of problems into simpler entities (objects) and allows for the modeling of real-world entities in software. OOP focuses on the following key concepts:

Key Concepts in OOP:

1. Class: A class is a blueprint or template that defines the data (attributes) and behaviors (methods) common to all objects of a certain kind.

2. Object: An object is an instance of a class. It is a concrete entity that exists based on the blueprint provided by the class.

3. Encapsulation: Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit (class).

4. Abstraction: Abstraction focuses on hiding complex implementation details and showing only the essential features of an object.

5. Inheritance: Inheritance is a mechanism by which one class (subclass or derived class) can inherit attributes and methods from another class (superclass or base class)..

6. Polymorphism: Polymorphism allows objects of different classes to be treated as instances of the same class through a common interface.

9.2. Class

A class is a blueprint or template for creating objects (instances). It defines the attributes (data) and methods (functions) that all instances of the class will have. Classes provide a way to logically group data and functions to facilitate reusability and modularity in code.

Key Concepts of Classes:

1. Attributes:

- Attributes are variables that hold data associated with a class and its objects.
- They represent the state or characteristics of objects created from the class.
- Example: In a Car class, attributes could include make, model, and year.

2. Methods:

- Methods are functions defined within a class that operate on objects created from the class.
- They define the behavior or actions that objects of the class can perform.
- Example: Methods in a Car class could include start(), drive(), and stop().

3. Constructor (__init__ method):

- The __init__ method is a special method used to initialize objects of a class.
- It is called automatically when an instance (object) of the class is created.
- The constructor initializes the object's attributes with the values passed during instantiation.
- Example: __init__(self, make, model, year) initializes the attributes make, model, and year when creating a Car object.

4. Instance Variables:

- Instance variables are variables that are unique to each instance (object) of a class.
- They hold data that is specific to each object.
- Example: self.make, self.model, and self.year are instance variables of a Car object.

5. Class Variables:

- Class variables are variables that are shared among all instances of a class.
- They are defined within the class but outside of any instance method.
- Example: A Car class might have a class variable num_wheels = 4 that applies to all instances (cars) of the class.

Example: Creating and Using a Class

Let's create a simple example of a Car class in Python to demonstrate its structure and usage:

1. Creating a Car Class

```
# Example: Creating a Car class
class Car:
    num_wheels = 4  # Class variable

    def __init__(self, make, model, year):
        self.make = make  # Instance variable
        self.model = model  # Instance variable
        self.year = year  # Instance variable

    def display_info(self):
        """Display information about the car."""
        print(f"{self.year} {self.make} {self.model} with {self.num_wheels}
wheels")
```

- **Class Definition:** class Car: begins the definition of the Car class.
- **Class Variable (num_wheels):** num_wheels = 4 is a class variable that is shared among all instances of the Car class.
- **Constructor (__init__ method):** __init__(self, make, model, year) is a special method used to initialize new objects of the Car class. It takes parameters make, model, and year to initialize the instance variables (self.make, self.model, self.year).

- **Instance Variables (make, model, year):** These are instance variables that store data specific to each instance of the Car class.
- **Instance Method (display_info()):** display_info(self) is a method defined within the Car class. It accesses instance variables (self.make, self.model, self.year) and a class variable (self.num_wheels) to display information about each car object.

2. Creating Objects and Accessing Methods

```
# Creating instances (objects) of the Car class
car1 = Car('Toyota', 'Camry', 2022)
car2 = Car('Honda', 'Accord', 2023)

# Accessing attributes and calling methods of objects
car1.display_info()
car2.display_info()

#Output:
2022 Toyota Camry with 4 wheels
2023 Honda Accord with 4 wheels
```

- **Object Creation:** car1 = Car('Toyota', 'Camry', 2022) and car2 = Car('Honda', 'Accord', 2023) create two instances (objects) of the Car class with specific attributes (make, model, year).
- **Method Invocation:** car1.display_info() and car2.display_info() invoke the display_info() method on each car object, which then accesses and prints the attributes (self.make, self.model, self.year) and the class variable (self.num_wheels).

9.3. Objects

An object is an instance of a class. It represents a unique entity with its own state (attributes) and behavior (methods), created based on the blueprint defined by the class.

Key Concepts of Objects:

1. Instance of a Class:

- An object is created from a class using the class constructor (__init__ method).
- Each object is independent and has its own set of attributes and methods.
- Example: If Car is a class, car1 and car2 are objects (instances) of the Car class.

2. Attributes:

- Attributes are variables that store data specific to each instance (object) of a class.
- They represent the state or characteristics of the object.
- Example: In a Car class, attributes could include make, model, and year.

3. Methods:

- Methods are functions defined within a class that define the object's behavior.
- They operate on the object's data (attributes) and perform actions related to the object.
- Example: Methods in a Car class could include start(), drive(), and stop().

4. Accessing Attributes and Calling Methods:

- Objects can access their attributes using dot notation (object.attribute).
- They can call methods defined in their class using dot notation (object.method()).
- Example: Creating and Using Objects

Let's continue with the example of the Car class from earlier to demonstrate how objects are created and used:

1. Creating a Class and Objects

```
# Example: Creating a Car class
class Car:
    num_wheels = 4  # Class variable

    def __init__(self, make, model, year):
        self.make = make  # Instance variable
        self.model = model  # Instance variable
        self.year = year  # Instance variable
```

- **Class Definition (Car):** Defines a blueprint for creating car objects.
- **Class Variable (num_wheels):** Shared by all instances of the class. In this case, it defines the number of wheels common to all cars.
- **Constructor (__init__):** Method called when creating a new instance of the class (Car). Initializes instance variables (self.make, self.model, self.year) with values passed during object creation.

2. Instantiating Objects

```
# Creating instances (objects) of the Car class
car1 = Car('Toyota', 'Camry', 2022)
car2 = Car('Honda', 'Accord', 2023)
```

- **Objects (car1, car2):** Instances of the Car class created using Car('Toyota', 'Camry', 2022) and Car('Honda', 'Accord', 2023), respectively. Each object (car1, car2) has its own unique attributes (make, model, year).

3. Accessing Attributes and Calling Methods

```
# Accessing attributes and calling methods of objects
print(f"Car 1: {car1.year} {car1.make} {car1.model}")
print(f"Car 2: {car2.year} {car2.make} {car2.model}")
```

- **Attribute Access:** car1.year, car1.make, car1.model access the attributes of car1.
- **Printing Attributes:** Displays the year, make, and model of each car object (car1, car2).

4. Defining and Calling Methods

```
# Defining a method within the Car class
def display_info(self):
    """Display information about the car."""
    print(f"{self.year} {self.make} {self.model} with {self.num_wheels}
wheels")

# Calling the method on objects
car1.display_info()
car2.display_info()
```

- **Method Definition (display_info):** Defines a method within the Car class to display information (year, make, model, num_wheels) about the car object.
- **Method Call:** car1.display_info() and car2.display_info() call the display_info() method on car1 and car2, respectively, to print information specific to each car object.

Real-World Example of Classes and Objects

To illustrate the concepts of classes and objects without focusing on encapsulation, let's consider a simple real-world example involving a Car class. We'll define a class to represent cars with attributes like make, model, and year, and methods to display information and start the car.

Example: Car Class

```
class Car:
    def __init__(self, make, model, year): #Initiate the variables
```

```
        self.make = make #Variable 1
        self.model = model #Variable 2
        self.year = year #Variable 3

    #  Methods to Display Information and Start the Car:
    def display_info(self):
        """Display information about the car."""
        print(f"Car Make: {self.make}")
        print(f"Car Model: {self.model}")
        print(f"Car Year: {self.year}")

    def start(self):
        """Simulate starting the car."""
        print(f"The {self.year} {self.make} {self.model} is starting.")
             print(

# Creating instances of the Car class
car1 = Car('Toyota', 'Corolla', 2020)
car2 = Car('Honda', 'Civic', 2019)

# Accessing attributes and calling methods
car1.display_info()
car1.start()

car2.display_info()
car2.start()
```

Output:

```
Car Make: Toyota
Car Model: Corolla
Car Year: 2020
The 2020 Toyota Corolla is starting.

Car Make: Honda
Car Model: Civic
Car Year: 2019
The 2019 Honda Civic is starting.
```

In this example, we've created a simple Car class with attributes for make, model, and year. We've defined methods to display information about the car and to simulate starting the car. We then created two instances of the Car class and demonstrated accessing their attributes and calling their methods. This example helps to illustrate the basic concepts of classes and objects in Python.

9.4. Encapsulation

- Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit called a class.
- It helps in achieving data hiding and abstraction, ensuring that the internal state of an object is accessible only through well-defined methods (getters and setters). This concept promotes modularity, security, and code reusability.
- In simple words, wrapping up an information in a variable.

Key Concepts of Encapsulation:

1. Data Hiding:

- **Public Access:** Attributes and methods that are accessible from outside the class without any restrictions.
- **Private Access:** Attributes and methods that are restricted to the class definition itself and not directly accessible from outside.
- **Protected Access:** Attributes and methods that are accessible within the class and its subclasses (inheritance hierarchy), providing a limited access scope.

2. Access Modifiers in Python:

- Python uses naming conventions with underscores ('_' and '__') to indicate the accessibility of attributes and methods.
- **Public:** Attributes/methods without underscore are conventionally considered public and can be accessed from outside the class.
- **Private (__):** Attributes/methods with a double underscore (__) are considered private and are not accessible directly from outside the class.
- **Protected (_):** Attributes/methods with a single underscore (_) at the beginning are considered protected, although this is more of a convention for indicating intended protected access.

1. Example: Public Access

In this example, we'll create a Student class with public attributes and methods. We'll then create an instance of this class and access its public attributes and methods.

1. Define the Student Class:

- Create a class named Student.
- Define the __init__ method (constructor) to initialize the object's attributes.

```
class Student:
    def __init__(self, name, student_id):
        self.name = name  # Public attribute
        self.student_id = student_id  # Public attribute
```

Explanation:

- Class Definition: class Student defines a new class named Student.
- Constructor Method: def __init__(self, name, student_id) is the constructor method that initializes the object's attributes when a new instance of the class is created.
- Public Attributes: self.name and self.student_id are public attributes. They can be accessed directly from outside the class.

2. Define a Method to Display Information:

Create a method named display_info within the Student class to print the student's information.

```
    def display_info(self):
        """Display information about the student."""
        print(f"Student Name: {self.name}")
        print(f"Student ID: {self.student_id}")
```

Explanation:

- Method Definition: def display_info(self) defines a method named display_info.
- Docstring: """Display information about the student.""" is a docstring describing what the method does.

- Print Statements: print(f"Student Name: {self.name}") and print(f"Student ID: {self.student_id}") print the student's name and ID, accessing the public attributes self.name and self.student_id.

3. Create an Instance of the Student Class:

Create an object (instance) of the Student class and initialize it with a name and student ID.

```
# Creating an instance of the Student class
student1 = Student('Alice', 'S001')
```

Explanation:

- Instance Creation: student1 = Student('Alice', 'S001') creates a new instance of the Student class with the name 'Alice' and student ID 'S001'. The __init__ method is called automatically, initializing student1.name to 'Alice' and student1.student_id to 'S001'.

4. Access Public Attributes and Call Methods:

Access the public attributes and call the method to display the student's information.

```
# Accessing public attributes and calling methods
print(f"Student Name: {student1.name}")
print(f"Student ID: {student1.student_id}")
student1.display_info()
```

Explanation:

- Accessing Public Attributes: print(f"Student Name: {student1.name}") and print(f"Student ID: {student1.student_id}") access and print the values of the public attributes name and student_id of the student1 object.
- Calling Method: student1.display_info() calls the display_info method of the student1 object, which prints the student's information.

Complete Code:

```
class Student:
    def __init__(self, name, student_id):
        self.name = name  # Public attribute
        self.student_id = student_id  # Public attribute
```

```
    def display_info(self):
        """Display information about the student."""
        print(f"Student Name: {self.name}")
        print(f"Student ID: {self.student_id}")

# Creating an instance of the Student class
student1 = Student('Alice', 'S001')

# Accessing public attributes and calling methods
print(f"Student Name: {student1.name}")
print(f"Student ID: {student1.student_id}")
student1.display_info()
```

Output:

```
Student Name: Alice
Student ID: S001
Student Name: Alice
Student ID: S001
```

2. Example: Private Access

In this example, we'll create a Department class with private attributes and methods. We'll demonstrate how to access these private attributes and methods using getter and setter methods.

1. Define the Department Class:

- Create a class named Department.
- Define the __init__ method (constructor) to initialize the object's attributes.

```
class Department:
    def __init__(self, name, code):
        self.name = name  # Public attribute
        self.__code = code  # Private attribute
```

Explanation:

- **Class Definition:** class Department defines a new class named Department.
- **Constructor Method:** def __init__(self, name, code) is the constructor method that initializes the object's attributes when a new instance of the class is created.
- **Public Attribute:** self.name is a public attribute that can be accessed directly from outside the class.
- **Private Attribute:** self.__code is a private attribute, indicated by the double underscore prefix (__). It is not accessible directly from outside the class.

2. Define Getter and Setter Methods for the Private Attribute:

Create methods to access and modify the private attribute __code.

```
def get_code(self):
    """Return the department code."""
    return self.__code

def set_code(self, code):
    """Set the department code."""
    if isinstance(code, str) and code:
        self.__code = code
    else:
        raise ValueError("Invalid code")
```

Explanation:

- **Getter Method:** def get_code(self) defines a method to return the value of the private attribute __code. This method allows controlled access to the private attribute.
- **Setter Method:** def set_code(self, code) defines a method to set the value of the private attribute __code. This method allows controlled modification of the private attribute. It includes validation to ensure the new code is a non-empty string.

3. Define a Method to Display Information:

Create a method named display_info within the Department class to print the department's information.

```
def display_info(self):
    """Display information about the department."""
```

```
        print(f"Department Name: {self.name}")
        print(f"Department Code: {self.__code}")
```

Explanation:

- **Method Definition:** def display_info(self) defines a method named display_info.
- **Docstring:** """Display information about the department.""" is a docstring describing what the method does.
- **Print Statements:** print(f"Department Name: {self.name}") and print(f"Department Code: {self.__code}") print the department's name and code, accessing the public attribute self.name and the private attribute self.__code.

4. Create an Instance of the Department Class:

Create an object (instance) of the Department class and initialize it with a name and department code.

```
# Creating an instance of the Department class
dept1 = Department('Human Resources', 'HR001')
```

Explanation:

- **Instance Creation:** dept1 = Department('Human Resources', 'HR001') creates a new instance of the Department class with the name 'Human Resources' and department code 'HR001'. The __init__ method is called automatically, initializing dept1.name to 'Human Resources' and dept1.__code to 'HR001'.

5. Access Private Attributes Using Getter and Setter Methods:

Use the getter and setter methods to access and modify the private attribute __code.

```
# Accessing and modifying private attribute using getter and setter methods
print(f"Department Code (using getter): {dept1.get_code()}")
dept1.set_code('HR002')
print(f"Updated Department Code (using getter): {dept1.get_code()}")
```

Explanation:

- **Getter Method:** print(f"Department Code (using getter): {dept1.get_code()}") calls the get_code method to access and print the value of the private attribute __code.

- **Setter Method:** dept1.set_code('HR002') calls the set_code method to modify the value of the private attribute __code to 'HR002'. The print(f"Updated Department Code (using getter): {dept1.get_code()}") statement calls the get_code method again to access and print the updated value of __code.

6. Display Information Using the Method:

Call the method to display the department's information.

```
# Calling the method to display department information
dept1.display_info()
```

Explanation:

- **Method Call:** dept1.display_info() calls the display_info method of the dept1 object, which prints the department's name and code.

Complete Code:

```
class Department:
    def __init__(self, name, code):
        self.name = name  # Public attribute
        self.__code = code  # Private attribute

    def get_code(self):
        """Return the department code."""
        return self.__code

    def set_code(self, code):
        """Set the department code."""
        if isinstance(code, str) and code:
            self.__code = code
        else:
            raise ValueError("Invalid code")

    def display_info(self):
        """Display information about the department."""
        print(f"Department Name: {self.name}")
        print(f"Department Code: {self.__code}")

# Creating an instance of the Department class
dept1 = Department('Human Resources', 'HR001')
```

```
# Accessing and modifying private attribute using getter and setter methods
print(f"Department Code (using getter): {dept1.get_code()}")
dept1.set_code('HR002')
print(f"Updated Department Code (using getter): {dept1.get_code()}")

# Calling the method to display department information
dept1.display_info()
```

Output:

```
Department Code (using getter): HR001
Updated Department Code (using getter): HR002
Department Name: Human Resources
Department Code: HR002
```

3. Example: Protected Access

In this example, we'll create a Student class with a protected attribute and demonstrate how it can be accessed within a derived class.

1. Define the Student Class:

- Create a class named Student.
- Define the __init__ method (constructor) to initialize the object's attributes.

```
class Student:
    def __init__(self, name, age):
        self.name = name  # Public attribute
        self._age = age  # Protected attribute
```

Explanation:

- **Class Definition:** class Student defines a new class named Student.
- **Constructor Method:** def __init__(self, name, age) is the constructor method that initializes the object's attributes when a new instance of the class is created.
- **Public Attribute:** self.name is a public attribute that can be accessed directly from outside the class.

- **Protected Attribute:** self._age is a protected attribute, indicated by the single underscore prefix (_). It is accessible within the class and its subclasses.

2. Define a Method to Display Information:

- Create a method named display_info within the Student class to print the student's information.

```
    def display_info(self):
        """Display information about the student."""
        print(f"Student Name: {self.name}")
        print(f"Student Age: {self._age}")
```

Explanation:

- **Method Definition:** def display_info(self) defines a method named display_info.
- **Docstring:** """Display information about the student.""" is a docstring describing what the method does.
- **Print Statements:** print(f"Student Name: {self.name}") and print(f"Student Age: {self._age}") print the student's name and age, accessing the public attribute self.name and the protected attribute self._age.

3. Define a Subclass to Access the Protected Attribute:

- Create a subclass named GraduateStudent that inherits from the Student class.
- Define the __init__ method to initialize additional attributes specific to GraduateStudent.

```
class GraduateStudent(Student):
    def __init__(self, name, age, degree):
        super().__init__(name, age)  # Initialize attributes from the
parent class
        self.degree = degree  # Public attribute
```

Explanation:

- **Subclass Definition:** class GraduateStudent(Student) defines a new class named GraduateStudent that inherits from the Student class.

- **Constructor Method:** def __init__(self, name, age, degree) is the constructor method that initializes the object's attributes.
- **Super Call:** super().__init__(name, age) calls the constructor of the parent Student class to initialize the name and _age attributes.
- **Additional Attribute:** self.degree is an additional public attribute specific to GraduateStudent.

4. Define a Method to Display Additional Information:

- Create a method named display_info within the GraduateStudent class to print the graduate student's information, including the protected attribute from the parent class.

```
    def display_info(self):
        """Display information about the graduate student."""
        super().display_info()  # Call the parent class method to display
name and age
        print(f"Degree: {self.degree}")
```

Explanation:

- **Method Definition:** def display_info(self) defines a method named display_info.
- **Docstring:** """Display information about the graduate student.""" is a docstring describing what the method does.
- **Super Call:** super().display_info() calls the display_info method of the parent Student class to display the name and age.
- **Print Statement:** print(f"Degree: {self.degree}") prints the graduate student's degree.

5. Create Instances and Demonstrate Access:

- Create objects (instances) of the Student and GraduateStudent classes.
- Call the methods to display their information.

```
# Creating an instance of the Student class
student1 = Student('Alice', 20)
student1.display_info()

# Creating an instance of the GraduateStudent class
```

```
grad_student1 = GraduateStudent('Bob', 25, 'MSc Computer Science')
grad_student1.display_info()
```

Explanation:

- Instance Creation: student1 = Student('Alice', 20) creates a new instance of the Student class with the name 'Alice' and age 20. grad_student1 = GraduateStudent('Bob', 25, 'MSc Computer Science') creates a new instance of the GraduateStudent class with the name 'Bob', age 25, and degree 'MSc Computer Science'.
- Method Calls: student1.display_info() and grad_student1.display_info() call the display_info methods to display the student's and graduate student's information, respectively.

Complete Code:

```
class Student:
    def __init__(self, name, age):
        self.name = name  # Public attribute
        self._age = age  # Protected attribute

    def display_info(self):
        """Display information about the student."""
        print(f"Student Name: {self.name}")
        print(f"Student Age: {self._age}")

class GraduateStudent(Student):
    def __init__(self, name, age, degree):
        super().__init__(name, age)  # Initialize attributes from the
parent class
        self.degree = degree  # Public attribute

    def display_info(self):
        """Display information about the graduate student."""
        super().display_info()  # Call the parent class method to display
name and age
        print(f"Degree: {self.degree}")

# Creating an instance of the Student class
student1 = Student('Alice', 20)
```

```
student1.display_info()

# Creating an instance of the GraduateStudent class
grad_student1 = GraduateStudent('Bob', 25, 'MSc Computer Science')
grad_student1.display_info()
```

Output:

```
Student Name: Alice
Student Age: 20
Student Name: Bob
Student Age: 25
Degree: MSc Computer Science
```

Real-World Example of Encapsulation in an E-commerce Department

Here is an example of a class that represents an e-commerce department. This example includes public, protected, and private attributes and methods to demonstrate encapsulation

```
class EcommerceDepartment:
    def __init__(self, name, manager, budget):
        self.name = name  # Public attribute
        self._manager = manager  # Protected attribute
        self.__budget = budget  # Private attribute

    def display_info(self):
        """Public method to display department information"""
        print(f"Department Name: {self.name}")  # Accessing public
attribute
        print(f"Manager: {self._manager}")  # Accessing protected attribute
        print(f"Budget: ${self.__budget}")  # Accessing private attribute

    def _get_manager(self):
        """Protected method to access the manager"""
        return self._manager

    def __adjust_budget(self, amount):
```

```
        """Private method to adjust the budget"""
        if isinstance(amount, (int, float)) and amount != 0:  # Validate
the adjustment amount
            self.__budget += amount  # Adjust the budget
        else:
            raise ValueError("Invalid budget adjustment amount")

    def adjust_budget(self, amount):
        """Public method to adjust budget with restricted access"""
        self.__adjust_budget(amount)  # Call private method to adjust
budget

# Creating an instance of the EcommerceDepartment class
dept = EcommerceDepartment('Electronics', 'Alice', 100000)

# Accessing public attribute and method
print(f"Public - Department Name: {dept.name}")  # Directly accessing
public attribute
dept.display_info()  # Calling public method to display department info

# Accessing protected attribute via a method
print(f"Protected - Manager: {dept._get_manager()}")  # Calling protected
method to access protected attribute

# Accessing private method via a public method
dept.adjust_budget(5000)  # Calling public method to adjust budget
print("After budget adjustment:")
dept.display_info()  # Calling public method again to display updated info
```

Output:

```
Public - Department Name: Electronics
Department Name: Electronics
Manager: Alice
Budget: $100000
Protected - Manager: Alice
After budget adjustment:
Department Name: Electronics
Manager: Alice
Budget: $105000
```

9.5. Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (known as a child or derived class) to inherit properties and behaviors (methods) from another class (known as a parent or base class). This mechanism promotes code reusability and establishes a natural hierarchical relationship between classes.

Key Concepts of Inheritance

- Parent Class (Base Class or Super Class): The class whose properties and methods are inherited.
- Child Class (Derived Class or Sub Class): The class that inherits properties and methods from the parent class.

Why Use Inheritance?

1. Code Reusability: Allows you to use existing code by inheriting properties and methods from the parent class.
2. Extend Functionality: Enables you to add new properties and methods to the child class without modifying the parent class.
3. Maintainability: Makes the code easier to maintain and update by organizing it into a hierarchy.

Types of Inheritance

1. Single Inheritance

- In single inheritance, a class inherits from one parent class.

2. Multiple Inheritance

- In multiple inheritance, a class can inherit from more than one parent class.

3. Multilevel Inheritance

- In multilevel inheritance, a class is derived from another derived class, forming a chain of inheritance.

4. Hierarchical Inheritance

- In hierarchical inheritance, multiple classes inherit from a single parent class.

5. Hybrid Inheritance

- Hybrid inheritance is a combination of two or more types of inheritance. It typically combines multiple and hierarchical inheritance.

1. Single Inheritance in Python

Single inheritance is a type of inheritance where a class (child class or derived class) inherits from only one parent class (base class or super class). This relationship allows the child class to inherit attributes and methods from the parent class, promoting code reusability and modularity.

Key Characteristics of Single Inheritance:

- **Parent Class:** Also known as base class or super class, it is the class whose attributes and methods are inherited by the child class.
- **Child Class:** Also known as derived class or sub class, it is the class that inherits properties and behaviors from the parent class

Syntax:

To create a child class that inherits from a parent class, use the following syntax:

```
class Parent:
```

```
    def parent_method(self):
        print("Parent Method")

class Child(Parent):
    def child_method(self):
        print("Child Method")

# Creating an instance of the Child class
child = Child()

# Accessing methods from both Parent and Child classes
child.parent_method()
child.child_method()
```

Example:

```
# Parent class representing father "Ramesh"
class Parent1:
    def speak(self):
        print("Ramesh says: Hello, I am the father.")

# Child class representing son "Vinod" inheriting from Parent1
class Child1(Parent1):
    def introduce(self):
        print("Vinod says: Hi, I am the son.")

# Driver code
son = Child1()
son.speak()    # Output: Ramesh says: Hello, I am the father.
son.introduce()    # Output: Vinod says: Hi, I am the son.
```

Explanation:

- Parent Class: Parent1 represents the father "Ramesh" with a method speak that prints a greeting.
- Child Class: Child1 represents the son "Vinod" inheriting from Parent1 and defines an additional method introduce that prints an introduction.
- Instance: son is an instance of Child1.
- Output: Calling son.speak() prints "Ramesh says: Hello, I am the father." and son.introduce() prints "Vinod says: Hi, I am the son."

Real-World Example: Animal and Dog

In this example, we'll model a simple inheritance hierarchy where Dog inherits from Animal.:

- Parent Class: Animal represents a generic animal with basic attributes and methods.
- Child Class: Dog inherits from Animal and adds specific attributes and methods relevant to a dog.

Example Code:

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

    def display_species(self):
        """Display the species of the animal."""
        print(f"This animal belongs to the species: {self.species}")

# Child class inheriting from Animal
class Dog(Animal):
    def __init__(self, species, breed):
        # Call the parent class constructor explicitly
        Animal.__init__(self, species)
        self.breed = breed

    def display_info(self):
        """Display information about the dog."""
        self.display_species()  # Call the parent class method
        print(f"This dog is of breed: {self.breed}")

# Creating instances
animal1 = Animal("Mammal")
dog1 = Dog("Mammal", "Labrador")

# Displaying information using methods from both classes
print("--- Animal Information ---")
animal1.display_species()

print("\n--- Dog Information ---")
```

```
dog1.display_info()
```

Output:

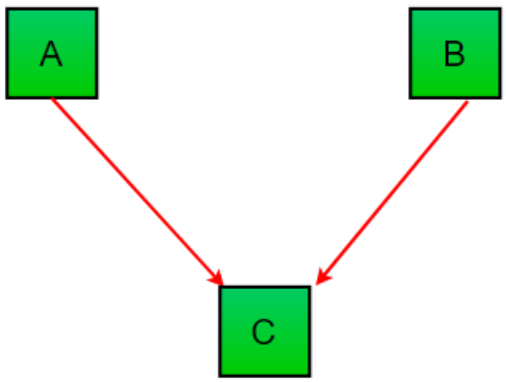
```
--- Animal Information ---
This animal belongs to the species: Mammal

--- Dog Information ---
This animal belongs to the species: Mammal
This dog is of breed: Labrador
```

Explanation:

- Animal Class: Represents a basic animal with the attribute species. The display_species method prints the species of the animal.
- Dog Class (Child Class): Inherits from Animal. It explicitly calls the constructor of Animal to initialize the species attribute. It adds an additional attribute breed and a method display_info that calls display_species to print the species of the dog and then prints the breed.

2. Multiple Inheritance:

Multiple inheritance is a type of inheritance where a class can inherit attributes and methods from more than one parent class. This allows the child class to combine features from multiple parent classes, promoting flexibility and code reuse.



Key Characteristics of Multiple Inheritance:

- **Parent Classes:** Multiple classes from which attributes and methods are inherited.
- **Child Class:** Class that inherits from multiple parent classes.

Syntax:

To create a child class that inherits from multiple parent classes, use the following syntax:

```
class Parent1:
    def method1(self):
        print("Method 1 from Parent1")

class Parent2:
    def method2(self):
        print("Method 2 from Parent2")

class Child(Parent1, Parent2):
    def child_method(self):
        print("Child Method")

# Creating an instance of the Child class
child = Child()

# Accessing methods from both Parent1 and Parent2 classes
child.method1()
child.method2()
child.child_method()
```

Example:

```
# Parent class 1
class Father:
    def speak(self):
        print("Father says: Hello, I am the father.")

# Parent class 2
class Mother:
    def care(self):
        print("Mother says: Hi, I am the mother.")
```

```
# Child class inheriting from both Father and Mother
class Child(Father, Mother):
    def introduce(self):
        print("Child says: Hi, I am their child.")

# Driver code
child = Child()
child.speak()      # Output: Father says: Hello, I am the father.
child.care()       # Output: Mother says: Hi, I am the mother.
child.introduce()  # Output: Child says: Hi, I am their child.
```

Explanation:

- **Parent Classes:**
  - Father: Represents the father with a method speak.
  - Mother: Represents the mother with a method care.
- **Child Class (Child):** Inherits from both Father and Mother.
  - introduce(): A method specific to Child.
- **Driver Code:** Creates an instance child of class Child.

Output:

```
child.speak(): Calls speak() from Father class.
child.care(): Calls care() from Mother class.
child.introduce(): Calls introduce() from Child class.
```

Real-World Example:

Consider an Employee class inheriting from both Person and Department classes. This example will show how an employee can have attributes from both a person and a department.

```
# Parent class 1
class Person:
    def __init__(self, name, age):
        # Initialize the attributes name and age for a person
        self.name = name
        self.age = age

    def display_person_info(self):
```

```
        # Display the person's name and age
        print(f"Name: {self.name}, Age: {self.age}")

# Parent class 2
class Department:
    def __init__(self, department_name):
        # Initialize the attribute department_name for a department
        self.department_name = department_name

    def display_department_info(self):
        # Display the department's name
        print(f"Department: {self.department_name}")

# Child class inheriting from both Person and Department
class Employee(Person, Department):
    def __init__(self, name, age, department_name, employee_id):
        # Initialize the attributes from Person class
        Person.__init__(self, name, age)
        # Initialize the attributes from Department class
        Department.__init__(self, department_name)
        # Initialize the attribute employee_id specific to Employee
        self.employee_id = employee_id

    def display_employee_info(self):
        # Call method from Person class to display person's information
        self.display_person_info()
        # Call method from Department class to display department's
information
        self.display_department_info()
        # Display the employee's ID
        print(f"Employee ID: {self.employee_id}")

# Driver code
# Create an instance of the Employee class
employee = Employee("Alice", 30, "Engineering", "E123")
# Display the employee's information by calling the method from the child
class
employee.display_employee_info()
```

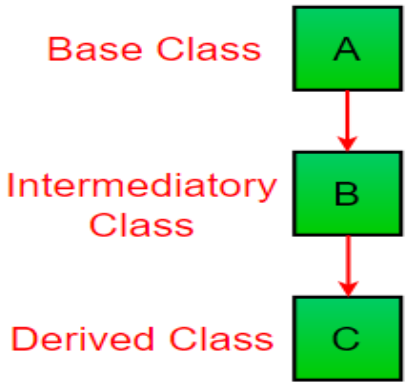
Output:

```
# Name: Alice, Age: 30
```

```
# Department: Engineering
# Employee ID: E123
```

3. Multilevel Inheritance in Python

Multilevel inheritance is a type of inheritance where a class (derived class) inherits from another class (intermediate base class), which itself is inherited from another class (base class). This creates a chain of inheritance where the derived class inherits properties and behaviors from multiple levels of parent classes.



Key Characteristics of Multilevel Inheritance:

- Base Class: The topmost class in the hierarchy.
- Intermediate Base Class: The class that inherits from the base class and acts as a parent class for the derived class.
- Derived Class: The class that inherits from the intermediate base class.

Syntax:

```
class Base:
    def base_method(self):
        print("Base Method")

class Intermediate(Base):
    def intermediate_method(self):
```

```
        print("Intermediate Method")

class Derived(Intermediate):
    def derived_method(self):
        print("Derived Method")

# Creating an instance of the Derived class
derived_instance = Derived()

# Accessing methods from Base, Intermediate, and Derived classes
derived_instance.base_method()
derived_instance.intermediate_method()
derived_instance.derived_method()
```

Example:

```
# Grandparent class
class Grandparent:
    def grandparent_method(self):
        print("This is the grandparent method.")

# Parent class inheriting from Grandparent
class Parent(Grandparent):
    def parent_method(self):
        print("This is the parent method.")

# Child class inheriting from Parent
class Child(Parent):
    def child_method(self):
        print("This is the child method.")

# Driver code
child = Child()
child.grandparent_method()  # Output: This is the grandparent method.
child.parent_method()       # Output: This is the parent method.
child.child_method()        # Output: This is the child method.
```

Real-World Example:

Consider a Company class, an ITDepartment class that inherits from Company, and a SoftwareEngineer class that inherits from ITDepartment. This example will show how attributes from all levels are combined.

```
# Base class representing a company
class Company:
    def __init__(self, company_name):
        # Initialize the company_name attribute
        self.company_name = company_name

    def display_company_info(self):
        # Display the company's name
        print(f"Company Name: {self.company_name}")

# Intermediate class representing the IT department, inheriting from
Company
class ITDepartment(Company):
    def __init__(self, company_name, department_head):
        # Initialize the attributes from Company class
        Company.__init__(self, company_name)
        # Initialize the department_head attribute
        self.department_head = department_head

    def display_it_department_info(self):
        # Call method from Company class to display company's information
        self.display_company_info()
        # Display the department head's name
        print(f"Department Head: {self.department_head}")

# Derived class representing a software engineer, inheriting from
ITDepartment
class SoftwareEngineer(ITDepartment):
    def __init__(self, company_name, department_head, employee_name,
skills):
        # Initialize the attributes from ITDepartment class
        ITDepartment.__init__(self, company_name, department_head)
        # Initialize the employee_name attribute
        self.employee_name = employee_name
        # Initialize the skills attribute
        self.skills = skills
```

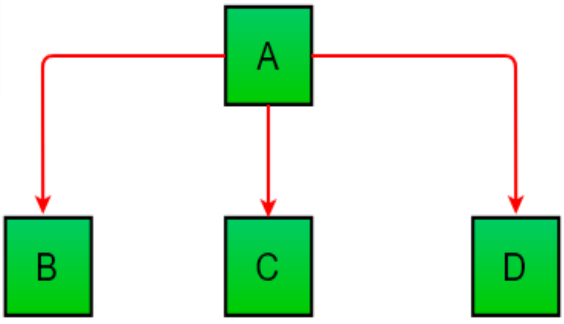
```
    def display_software_engineer_info(self):
        # Call method from ITDepartment class to display IT department's
information
        self.display_it_department_info()
        # Display the employee's name
        print(f"Employee Name: {self.employee_name}")
        # Display the employee's skills
        print(f"Skills: {', '.join(self.skills)}")

# Driver code
# Create an instance of the SoftwareEngineer class
engineer = SoftwareEngineer("TechCorp", "Alice Smith", "John Doe",
["Python", "Java", "C++"])
# Display the software engineer's information by calling the method from
the child class
engineer.display_software_engineer_info()
```

Output:

```
Company Name: TechCorp
Department Head: Alice Smith
Employee Name: John Doe
Skills: Python, Java, C++
```

4. Hierarchical Inheritance

Hierarchical inheritance occurs when multiple classes inherit from a single base class. This means that a single parent class can have multiple child classes, each inheriting the attributes and methods of the parent class.



Key Characteristics of Hierarchical Inheritance:

- **Base Class (Parent Class):** The single class from which multiple child classes inherit.
- **Derived Classes (Child Classes):** Multiple classes that inherit from the base class, each potentially having its own attributes and methods.

Syntax:

```
class Base:
    def base_method(self):
        print("Base Method")

class Derived1(Base):
    def derived1_method(self):
        print("Derived1 Method")

class Derived2(Base):
    def derived2_method(self):
        print("Derived2 Method")

# Creating instances of the derived classes
derived1 = Derived1()
derived2 = Derived2()

# Accessing methods from the base class and derived classes
derived1.base_method()
derived1.derived1_method()
derived2.base_method()
derived2.derived2_method()
```

Example:

Let's take a simple example where we have a Person class, and two classes Student and Teacher that inherit from Person

```
# Base class representing a person
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    def display_person_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Derived class representing a student, inheriting from Person
class Student(Person):
    def __init__(self, name, age, student_id):
        Person.__init__(self, name, age)  # Initialize Person attributes
        self.student_id = student_id

    def display_student_info(self):
        self.display_person_info()  # Call method from Person
        print(f"Student ID: {self.student_id}")

# Derived class representing a teacher, inheriting from Person
class Teacher(Person):
    def __init__(self, name, age, employee_id):
        Person.__init__(self, name, age)  # Initialize Person attributes
        self.employee_id = employee_id

    def display_teacher_info(self):
        self.display_person_info()  # Call method from Person
        print(f"Employee ID: {self.employee_id}")

# Driver code
student = Student("John Doe", 20, "S12345")
teacher = Teacher("Jane Smith", 35, "T98765")

student.display_student_info()
teacher.display_teacher_info()
```

Output:

```
Name: John Doe, Age: 20
Student ID: S12345
Name: Jane Smith, Age: 35
Employee ID: T98765
```

Real-World Example

Consider a Company class, and two classes HRDepartment and EngineeringDepartment that inherit from Company. This example will show how attributes from the base class are shared among multiple derived classes.

```
# Base class representing a company
class Company:
    def __init__(self, company_name):
        # Initialize the company_name attribute
        self.company_name = company_name

    def display_company_info(self):
        # Display the company's name
        print(f"Company Name: {self.company_name}")

# Derived class representing the HR department, inheriting from Company
class HRDepartment(Company):
    def __init__(self, company_name, hr_head):
        # Initialize the attributes from Company class
        Company.__init__(self, company_name)
        # Initialize the hr_head attribute
        self.hr_head = hr_head

    def display_hr_info(self):
        # Call method from Company class to display company's information
        self.display_company_info()
        # Display the HR head's name
        print(f"HR Head: {self.hr_head}")

# Derived class representing the Engineering department, inheriting from
Company
class EngineeringDepartment(Company):
    def __init__(self, company_name, engineering_head):
        # Initialize the attributes from Company class
        Company.__init__(self, company_name)
        # Initialize the engineering_head attribute
        self.engineering_head = engineering_head

    def display_engineering_info(self):
        # Call method from Company class to display company's information
        self.display_company_info()
        # Display the engineering head's name
```

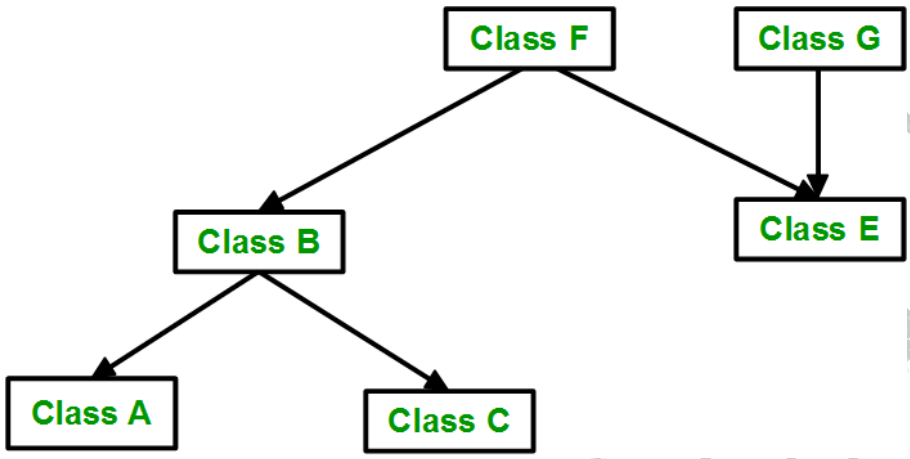
```
        print(f"Engineering Head: {self.engineering_head}")

# Driver code
# Create an instance of the HRDepartment class
hr_department = HRDepartment("TechCorp", "Alice Smith")
# Create an instance of the EngineeringDepartment class
engineering_department = EngineeringDepartment("TechCorp", "Bob Johnson")

# Display the HR department's information by calling the method from the
derived class
hr_department.display_hr_info()
# Display the Engineering department's information by calling the method
from the derived class
engineering_department.display_engineering_info()
```

Output:

```
Company Name: TechCorp
HR Head: Alice Smith
Company Name: TechCorp
Engineering Head: Bob Johnson
```

5. Hybrid Inheritance

Hybrid inheritance combines multiple types of inheritance, such as single, multiple, and hierarchical inheritance. This allows classes to inherit from multiple base classes, creating a complex inheritance hierarchy.



Key Characteristics of Hybrid Inheritance:

- **Multiple Base Classes:** A derived class can inherit from multiple base classes, including single, multiple, and hierarchical inheritance.
- **Combination of Inheritance Types:** Allows for flexible reuse of code and modeling of complex relationships between classes.

Syntax:

```
class Base1:
    def method_base1(self):
        print("Base1 Method")

class Base2:
    def method_base2(self):
        print("Base2 Method")

class Derived(Base1, Base2):
    def method_derived(self):
        print("Derived Method")

# Creating an instance of the derived class
derived = Derived()

# Accessing methods from both base classes and the derived class
derived.method_base1()
derived.method_base2()
derived.method_derived()
```

Example:

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

    def display_person_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Derived class from Person (Single Inheritance)
class Student(Person):
    def __init__(self, name, age, student_id):
        Person.__init__(self, name, age)
        self.student_id = student_id

    def display_student_info(self):
        self.display_person_info()
        print(f"Student ID: {self.student_id}")

# Another base class
class Employee:
    def __init__(self, employee_id):
        self.employee_id = employee_id

    def display_employee_info(self):
        print(f"Employee ID: {self.employee_id}")

# Derived class from both Student and Employee (Multiple Inheritance)
class WorkingStudent(Student, Employee):
    def __init__(self, name, age, student_id, employee_id):
        Student.__init__(self, name, age, student_id)
        Employee.__init__(self, employee_id)

    def display_working_student_info(self):
        self.display_person_info()
        self.display_student_info()
        self.display_employee_info()

# Driver code
working_student = WorkingStudent("Alice", 22, "S12345", "E98765")
```

```
working_student.display_working_student_info()
```

Output:

```
Name: Alice, Age: 22
Name: Alice, Age: 22
Student ID: S12345
Employee ID: E98765
```

Real-World Example

Consider a company where employees can also be students (e.g., interns). We will create a hybrid inheritance structure to represent this.

```
# Base class representing a person
class Person:
    def __init__(self, name, age):
        # Initialize name and age attributes
        self.name = name
        self.age = age

    def display_person_info(self):
        # Display the person's name and age
        print(f"Name: {self.name}, Age: {self.age}")

# Derived class representing a student, inheriting from Person
class Student(Person):
    def __init__(self, name, age, student_id):
        # Initialize attributes from Person class
        Person.__init__(self, name, age)
        # Initialize student_id attribute
        self.student_id = student_id

    def display_student_info(self):
        # Call method from Person class to display person's information
        self.display_person_info()
        # Display the student's ID
        print(f"Student ID: {self.student_id}")

# Another base class representing an employee
class Employee:
    def __init__(self, employee_id):
```

```
        # Initialize employee_id attribute
        self.employee_id = employee_id

    def display_employee_info(self):
        # Display the employee's ID
        print(f"Employee ID: {self.employee_id}")

# Derived class representing a working student, inheriting from both
Student and Employee
class WorkingStudent(Student, Employee):
    def __init__(self, name, age, student_id, employee_id):
        # Initialize attributes from Student class
        Student.__init__(self, name, age, student_id)
        # Initialize attributes from Employee class
        Employee.__init__(self, employee_id)

    def display_working_student_info(self):
        # Call methods from Student and Employee classes to display their
information
        self.display_person_info()
        self.display_student_info()
        self.display_employee_info()

# Driver code
# Create an instance of the WorkingStudent class
working_student = WorkingStudent("John Doe", 21, "S67890", "E54321")

# Display the working student's information by calling the method from the
derived class
working_student.display_working_student_info()
```

Output:

```
Name: John Doe, Age: 21
Name: John Doe, Age: 21
Student ID: S67890
Employee ID: E54321
```

Problem Statement 1:

Create a class hierarchy for animals, including a base class Animal and two derived classes Dog and Cat. Implement methods for each class to display their attributes and details.

Code:

```
# Base class representing an Animal
class Animal:
    def __init__(self, species):
        self.species = species

    def display_species(self):
        print(f"Species: {self.species}")

# Derived class representing a Dog, inheriting from Animal
class Dog(Animal):
    def __init__(self, species, breed):
        Animal.__init__(self, species)
        self.breed = breed

    def display_details(self):
        self.display_species()
        print(f"Breed: {self.breed}")
        print("Type: Dog")

# Derived class representing a Cat, inheriting from Animal
class Cat(Animal):
    def __init__(self, species, color):
        Animal.__init__(self, species)
        self.color = color

    def display_details(self):
        self.display_species()
        print(f"Color: {self.color}")
        print("Type: Cat")

# Driver code
dog = Dog("Canine", "Labrador Retriever")
cat = Cat("Feline", "Tabby")

print("--- Dog Details ---")
dog.display_details()
```

```
print("\n--- Cat Details ---")
cat.display_details()
```

Output:

```
--- Dog Details ---
Species: Canine
Breed: Labrador Retriever
Type: Dog

--- Cat Details ---
Species: Feline
Color: Tabby
Type: Cat
```

Explanation:

- Animal Class: Base class with an attribute species and a method display_species to print the species.
- Dog Class: Inherits from Animal. Adds an additional attribute breed. Initializes species using Animal.__init__(self, species) and includes breed in display_details.
- Cat Class: Inherits from Animal. Adds an additional attribute color. Initializes species using Animal.__init__(self, species) and includes color in display_details.
- Driver Code: Creates instances of Dog and Cat, initializes their attributes, and displays their details using the display_details method.

Problem Statement 2:

Create a class hierarchy for electronic devices, including a base class Device and two derived classes Phone and Laptop. Implement methods for each class to display their attributes and details. Use multiple inheritance to introduce an additional feature class Camera that both Phone and Laptop can inherit from.

Code:

```
# Base class representing a Device
class Device:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

    def display_info(self):
        print(f"Brand: {self.brand}")
        print(f"Year: {self.year}")

# Feature class representing a Camera
class Camera:
    def __init__(self, resolution):
        self.resolution = resolution

    def display_camera_info(self):
        print(f"Camera Resolution: {self.resolution}")

# Derived class representing a Phone, inheriting from Device and Camera
class Phone(Device, Camera):
    def __init__(self, brand, year, model, resolution):
        Device.__init__(self, brand, year)
        Camera.__init__(self, resolution)
        self.model = model

    def display_details(self):
        self.display_info()
        print(f"Model: {self.model}")
        self.display_camera_info()
        print("Type: Phone")

# Derived class representing a Laptop, inheriting from Device and Camera
class Laptop(Device, Camera):
    def __init__(self, brand, year, model, resolution):
        Device.__init__(self, brand, year)
        Camera.__init__(self, resolution)
        self.model = model

    def display_details(self):
        self.display_info()
        print(f"Model: {self.model}")
        self.display_camera_info()
```

```
        print("Type: Laptop")

# Driver code
phone = Phone("Apple", 2022, "iPhone 14", "12 MP")
laptop = Laptop("Dell", 2021, "XPS 15", "720p")

print("--- Phone Details ---")
phone.display_details()

print("\n--- Laptop Details ---")
laptop.display_details()
```

Explanation:

- Device Class: Base class with attributes brand and year. Includes a method display_info to print brand and year.
- Camera Class: Feature class with an attribute resolution and a method display_camera_info to print camera resolution.
- Phone Class: Inherits from Device and Camera. Adds an additional attribute model. Initializes brand, year, and resolution using Device.__init__(self, brand, year) and Camera.__init__(self, resolution) and includes model in display_details.
- Laptop Class: Inherits from Device and Camera. Adds an additional attribute model. Initializes brand, year, and resolution using Device.__init__(self, brand, year) and Camera.__init__(self, resolution) and includes model in display_details.
- Driver Code: Creates instances of Phone and Laptop, initializes their attributes, and displays their details using the display_details method.

Output:

```
--- Phone Details ---
Brand: Apple
Year: 2022
Model: iPhone 14
Camera Resolution: 12 MP
Type: Phone

--- Laptop Details ---
Brand: Dell
Year: 2021
```

```
Model: XPS 15
Camera Resolution: 720p
Type: Laptop
```

9.6. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different underlying forms (data types). In Python, polymorphism is commonly achieved through method overriding and method overloading.

Key Characteristics of Polymorphism:

1. **Method Overriding:** Allows a child class to provide a specific implementation of a method that is already defined in its parent class.
2. **Method Overloading:** Allows multiple methods in the same class with the same name but different parameters. Note that Python does not support traditional method overloading, but similar behavior can be achieved using default arguments.

Syntax:

```
class Parent:
    def show(self):
        print("This is the parent class method")

class Child(Parent):
    def show(self):
        print("This is the child class method")

# Creating an instance of Child class
child = Child()
child.show()  # Output: This is the child class method
```

Method Overloading Example:

```
class Calculator:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
```

```
        return a + b + c

# Creating an instance of Calculator
calc = Calculator()

# Method Overloading Example
print(calc.add(2, 3))       # This will raise an error because the add
method is overloaded
print(calc.add(2, 3, 4))    # This will print 9
```

Explanation of Code:

- Calculator Class:
  - Defines a class Calculator.
- add Method:
  - def add(self, a, b): Defines the add method with two parameters a and b, which adds two numbers.
  - def add(self, a, b, c): Re-defines the add method with three parameters a, b, and c, which adds three numbers.
- Creating Instance:
  - calc = Calculator(): Creates an instance calc of the Calculator class.
- Method Overloading Example:
  - print(calc.add(2, 3)): Attempts to call the add method with two arguments. This will raise an error because Python does not support traditional method overloading where multiple methods with the same name but different signatures can coexist.
  - print(calc.add(2, 3, 4)): Calls the add method with three arguments, demonstrating method overloading by defining a method with the same name but different parameters.

Method Overriding Example:

```
class Animal:
    def make_sound(self):
        print("Generic animal sound")

class Dog(Animal):
```

```
    def make_sound(self):
        print("Bark")

# Creating instances
animal = Animal()
dog = Dog()

# Method Overriding Example
animal.make_sound()  # Output: Generic animal sound
dog.make_sound()     # Output: Bark
```

Explanation:

- Animal Class:
  - Defines a class Animal.
  - def make_sound(self): Defines the method make_sound that prints a generic animal sound.
- Dog Class (Inherits from Animal):
  - class Dog(Animal): Defines a subclass Dog that inherits from Animal.
  - def make_sound(self): Overrides the make_sound method of Animal to print "Bark" instead of the generic animal sound.
- Creating Instances:
  - animal = Animal(): Creates an instance animal of the Animal class.
  - dog = Dog(): Creates an instance dog of the Dog class.
- Method Overriding Example:
  - animal.make_sound(): Calls the make_sound method on the animal instance. Since Animal does not override make_sound, it prints "Generic animal sound".
  - dog.make_sound(): Calls the make_sound method on the dog instance. This demonstrates method overriding where the Dog class provides its own implementation of make_sound, printing "Bark" instead of the generic sound.

Real-World Example:

Consider a scenario where we have different types of vehicles. We want to create a common interface for them to display their type of fuel.

```
# Parent class
class Vehicle:
    def __init__(self, make, model):
        # Initialize common attributes make and model
        self.make = make
        self.model = model

    def fuel_type(self):
        # Placeholder method to be overridden by child classes
        pass

# Child class 1
class PetrolVehicle(Vehicle):
    def fuel_type(self):
        # Override the fuel_type method to specify petrol as the fuel
        print(f"The {self.make} {self.model} runs on petrol")

# Child class 2
class DieselVehicle(Vehicle):
    def fuel_type(self):
        # Override the fuel_type method to specify diesel as the fuel
        print(f"The {self.make} {self.model} runs on diesel")

# Child class 3
class ElectricVehicle(Vehicle):
    def fuel_type(self):
        # Override the fuel_type method to specify electricity as the fuel
        print(f"The {self.make} {self.model} runs on electricity")

# Creating instances of different vehicles
car = PetrolVehicle("Toyota", "Corolla")
truck = DieselVehicle("Ford", "F-150")
bike = ElectricVehicle("Tesla", "Model S")

# Demonstrating polymorphism
vehicles = [car, truck, bike]
for vehicle in vehicles:
    vehicle.fuel_type()  # Output: Fuel type of each vehicle
```

Output:

```
The Toyota Corolla runs on petrol
```

```
The Ford F-150 runs on diesel
The Tesla Model S runs on electricity
```

Explanation:

- Vehicle Class: The parent class with a placeholder method fuel_type that is meant to be overridden by child classes.
- PetrolVehicle, DieselVehicle, ElectricVehicle Classes: Child classes that inherit from Vehicle and override the fuel_type method to specify their respective fuel types.
- Instances: Instances of PetrolVehicle, DieselVehicle, and ElectricVehicle are created with specific makes and models.
- Polymorphism Demonstration: A list of vehicles is iterated over, and the fuel_type method is called on each, demonstrating polymorphism where the same method name results in different behaviors based on the object type.

Problem Statement 1: Shapes and Area Calculation

Create a program that demonstrates polymorphism through different types of shapes calculating their respective areas. Define a parent class Shape with a method area, and child classes Circle and Rectangle that override this method to calculate and print their respective areas.

Code:

```
import math

# Parent class
class Shape:
    def area(self):
        # Placeholder method to be overridden by child classes
        pass

# Child class 1
class Circle(Shape):
    def __init__(self, radius):
        # Initialize attribute radius
        self.radius = radius
```

```
    def area(self):
        # Override area method to calculate area of circle
        print(f"Area of Circle: {math.pi * self.radius ** 2}")

# Child class 2
class Rectangle(Shape):
    def __init__(self, length, width):
        # Initialize attributes length and width
        self.length = length
        self.width = width

    def area(self):
        # Override area method to calculate area of rectangle
        print(f"Area of Rectangle: {self.length * self.width}")

# Creating instances of different shapes
circle = Circle(5)
rectangle = Rectangle(4, 6)

# Demonstrating polymorphism
shapes = [circle, rectangle]
for shape in shapes:
    shape.area()  # Output: Area of each shape
```

Output:

```
Area of Circle: 78.53981633974483
Area of Rectangle: 24
```

Problem Statement 2: Employees and their Salaries

Create a program that demonstrates polymorphism through different types of employees calculating their respective salaries. Define a parent class Employee with a method calculate_salary, and child classes FullTimeEmployee and PartTimeEmployee that override this method to calculate and print their respective salaries.

Code:

```
# Parent class
```

```
class Employee:
    def calculate_salary(self):
        # Placeholder method to be overridden by child classes
        pass

# Child class 1
class FullTimeEmployee(Employee):
    def __init__(self, base_salary, bonus):
        # Initialize attributes base_salary and bonus
        self.base_salary = base_salary
        self.bonus = bonus

    def calculate_salary(self):
        # Override calculate_salary method to calculate salary of full-time
employee
        print(f"Full-Time Employee Salary: {self.base_salary +
self.bonus}")

# Child class 2
class PartTimeEmployee(Employee):
    def __init__(self, hourly_rate, hours_worked):
        # Initialize attributes hourly_rate and hours_worked
        self.hourly_rate = hourly_rate
        self.hours_worked = hours_worked

    def calculate_salary(self):
        # Override calculate_salary method to calculate salary of part-time
employee
        print(f"Part-Time Employee Salary: {self.hourly_rate *
self.hours_worked}")

# Creating instances of different employees
full_time_employee = FullTimeEmployee(50000, 10000)
part_time_employee = PartTimeEmployee(20, 100)

# Demonstrating polymorphism
employees = [full_time_employee, part_time_employee]
for employee in employees:
    employee.calculate_salary()  # Output: Salary of each employee
```

Output:

```
Full-Time Employee Salary: 60000
Part-Time Employee Salary: 2000
```

Explanation:

- Employee Class: The parent class with a placeholder method calculate_salary.
- FullTimeEmployee Class: A child class that inherits from Employee, initializes base_salary and bonus, and overrides the calculate_salary method to calculate the salary of a full-time employee.
- PartTimeEmployee Class: A child class that inherits from Employee, initializes hourly_rate and hours_worked, and overrides the calculate_salary method to calculate the salary of a part-time employee.
- Instances: Instances of FullTimeEmployee and PartTimeEmployee are created with specific salary parameters.
- Polymorphism Demonstration: A list of employees is iterated over, and the calculate_salary method is called on each, demonstrating polymorphism where the same method name results in different behaviors based on the object type.

9.7. Abstraction

Abstraction is the process of hiding the complex implementation details and exposing only the essential features of an object or a class. It focuses on the 'what' rather than the 'how'. In other words, abstraction allows us to define a blueprint for creating objects without getting into the specifics of how each part works internally.

Key Concepts in Abstraction

1. Abstract Class:

- An abstract class is a blueprint for other classes and cannot be instantiated on its own.
- It typically contains one or more abstract methods, which are declared but contain no implementation.
- Abstract classes may also include concrete methods (methods with implementations) that can be used by its subclasses.

2. Abstract Method:

- An abstract method is a method declared in an abstract class but lacks implementation.
- Subclasses of an abstract class must provide concrete implementations for all abstract methods unless they themselves are abstract.

3. Interfaces:

- Iinterfaces are typically represented using abstract base classes (ABCs) from the abc module.
- Interfaces define a set of methods that must be implemented by any class that inherits from them.
- They help enforce a contract where classes provide specific behaviors, ensuring consistency in application design.

1. Abstract Class:

An abstract class that cannot be instantiated directly. It serves as a blueprint for other classes and typically contains one or more abstract methods, which are declared but not implemented in the abstract class itself. Abstract classes may also include concrete methods with implementations that can be used by its subclasses.

Syntax:

```
from abc import ABC, abstractmethod

class AbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass

    def concrete_method(self):
        print("This is a concrete method in the abstract class.")
```

Explanation of Syntax:

- ABC and abstractmethod from abc module: ABC is a metaclass that allows defining abstract classes, and abstractmethod is a decorator used to declare abstract methods.
- AbstractClass Definition:
    - Inherits from ABC, marking it as an abstract class.
    - Contains an abstract method abstract_method() declared with @abstractmethod but not implemented (pass statement).
    - Includes a concrete method concrete_method() that provides a default implementation

Example:

```
from abc import ABC, abstractmethod

# Abstract class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

```
    @abstractmethod
    def perimeter(self):
        pass

# Concrete class Circle implementing Shape
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14 * self.radius

# Creating an instance of Circle
circle = Circle(5)

# Calling methods from the Circle instance
print("Circle Area:", circle.area())         # Output: Circle Area: 78.5
print("Circle Perimeter:", circle.perimeter())   # Output: Circle
Perimeter: 31.400000000000002
```

Explanation of Example:

- Shape Abstract Class:
  - Defines two abstract methods area() and perimeter() which must be implemented by any concrete subclass (Circle in this case).
- Circle Concrete Class:
  - Inherits from Shape and provides concrete implementations for area() and perimeter().
  - Calculates the area and perimeter of a circle using the given formulas.
- Usage:
  - Creates an instance circle of type Circle with a radius of 5.
  - Calls area() and perimeter() methods on circle, which executes the implementations specific to the Circle class.

Real-World Example:

Consider a scenario where you have a base class Vehicle that defines an abstract method start(). This abstract class is inherited by concrete classes like Car, Bus, and Truck, each implementing its own version of start() method based on their specific functionalities (e.g., engine start for Car, ignition for Bus, etc.).

```
from abc import ABC, abstractmethod

# Abstract class
class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass

# Concrete class Car implementing Vehicle
class Car(Vehicle):
    def start(self):
        print("Car engine started.")

# Concrete class Bus implementing Vehicle
class Bus(Vehicle):
    def start(self):
        print("Bus ignition activated.")

# Creating instances of different vehicles
car = Car()
bus = Bus()

# Starting each vehicle
car.start()   # Output: Car engine started.
bus.start()   # Output: Bus ignition activated.
```

Explanation of Real-World Example:

- Vehicle Abstract Class:
  - Declares an abstract method start() which defines a common interface for starting different types of vehicles.
- Car and Bus Concrete Classes:
  - Inherit from Vehicle and provide specific implementations for start() tailored to their respective vehicle types.

- Usage:
    - Creates instances car and bus of types Car and Bus.
    - Calls start() method on each instance, resulting in vehicle-specific actions (Car engine started. and Bus ignition activated.).

2. Abstract Method:

Abstract methods are methods defined in abstract classes that have no implementation in the abstract class itself. Instead, their implementation is left to be defined by subclasses that inherit from the abstract class. Abstract methods serve as a blueprint for methods that must be implemented by any subclass, ensuring a consistent interface across different subclasses.

Syntax:

```
from abc import ABC, abstractmethod

class AbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass
```

- ABC and abstractmethod from abc module:
    - ABC is a metaclass used to define abstract classes.
    - abstractmethod is a decorator that marks a method as abstract, indicating it must be implemented by any subclass of the abstract class.
- AbstractClass Definition:
    - Inherits from ABC, marking it as an abstract class.
    - Contains an abstract method abstract_method() declared with @abstractmethod but not implemented (pass statement).

Purpose of Abstract Methods:

1. **Enforce Method Contracts:** Abstract methods enforce a contract that specifies methods subclasses must implement, ensuring consistency and conformity to a certain interface.
2. **Provide Structure:** They provide a structural framework for defining methods that subclasses are expected to implement, promoting code organization and clarity.

Example:

```
from abc import ABC, abstractmethod

# Abstract class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

# Concrete class Circle implementing Shape
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

# Creating an instance of Circle
circle = Circle(5)

# Calling the area method
print("Area of Circle:", circle.area())   # Output: Area of Circle: 78.5
```

Explanation:

- Shape Abstract Class:
  - Declares an abstract method area() using @abstractmethod, which must be implemented by any subclass (Circle in this case).
- Circle Concrete Class:
  - Inherits from Shape and provides a concrete implementation for the area() method, calculating the area of a circle based on its radius.
- Usage:

- Creates an instance circle of type Circle with a radius of 5.
- Calls the area() method on circle, which executes the implementation specific to the Circle class, returning the area of the circle.

Real-World Example:

Consider an abstract class Animal with an abstract method make_sound(). Concrete subclasses like Dog and Cat inherit from Animal and provide their own implementations of make_sound(), representing the distinct sounds each animal makes.

```
from abc import ABC, abstractmethod

# Abstract class
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

# Concrete class Dog implementing Animal
class Dog(Animal):
    def make_sound(self):
        print("Woof!")

# Concrete class Cat implementing Animal
class Cat(Animal):
    def make_sound(self):
        print("Meow!")

# Creating instances of different animals
dog = Dog()
cat = Cat()

# Calling make_sound method on each animal
dog.make_sound()   # Output: Woof!
cat.make_sound()   # Output: Meow!
```

Explanation:

- Animal Abstract Class:

    - Defines an abstract method make_sound() using @abstractmethod, specifying that subclasses (Dog and Cat) must implement this method.
- Dog and Cat Concrete Classes:
    - Inherit from Animal and provide specific implementations for make_sound(), representing the distinct sounds made by dogs and cats.
- Usage:
    - Creates instances dog and cat of types Dog and Cat.
    - Calls the make_sound() method on each instance, executing the implementation specific to each subclass (Woof! for Dog and Meow! for Cat).

3. Interfaces

While Python does not have a formal concept of interfaces as in languages like Java or C#, you can achieve similar functionality using abstract classes. In Python, an interface can be represented by an abstract class that only contains abstract methods. The subclasses then implement these abstract methods.

- An interface is a class with one or more abstract methods. These methods do not have implementations in the interface itself but are implemented by the classes that inherit from the interface.
- Interfaces define a contract that classes must adhere to, ensuring that they implement specific methods. This allows different classes to be used interchangeably if they implement the same interface, promoting polymorphism and loose coupling in code.

Syntax:

```
from abc import ABC, abstractmethod

class Interface(ABC):
    @abstractmethod
    def method1(self):
        pass
```

```
    @abstractmethod
    def method2(self):
        pass
```

Explanation of Syntax:

- ABC and abstractmethod from abc module:
    - ABC is a metaclass used to define abstract classes (and hence interfaces).
    - abstractmethod is a decorator that marks methods as abstract, indicating they must be implemented by any subclass.
- Interface Definition:
    - Inherits from ABC, marking it as an abstract class.
    - Contains abstract methods method1() and method2() declared with @abstractmethod but not implemented (pass statement).

Example:

```
from abc import ABC, abstractmethod

# Interface class
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

    @abstractmethod
    def move(self):
        pass

# Concrete class Dog implementing Animal interface
class Dog(Animal):
    def make_sound(self):
        print("Woof!")

    def move(self):
        print("Dog is running")

# Concrete class Cat implementing Animal interface
class Cat(Animal):
    def make_sound(self):
```

```
        print("Meow!")

    def move(self):
        print("Cat is walking")

# Creating instances of different animals
dog = Dog()
cat = Cat()

# Using the interface methods
dog.make_sound()  # Output: Woof!
dog.move()        # Output: Dog is running
cat.make_sound()  # Output: Meow!
cat.move()        # Output: Cat is walking
```

Explanation of Example:

- Animal Interface:
  - Defines two abstract methods: make_sound() and move().
- Dog and Cat Concrete Classes:
  - Inherit from Animal and provide specific implementations for make_sound() and move(), representing behaviors specific to dogs and cats.
- Usage:
  - Creates instances dog and cat of types Dog and Cat.
  - Calls the make_sound() and move() methods on each instance, executing the implementation specific to each subclass.

Real-World Example:

Consider an interface PaymentProcessor with abstract methods process_payment() and refund_payment(). Concrete subclasses like CreditCardProcessor and PayPalProcessor inherit from PaymentProcessor and provide their own implementations of these methods.

```
from abc import ABC, abstractmethod

# Interface class
class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount):
```

```
        pass

    @abstractmethod
    def refund_payment(self, amount):
        pass

# Concrete class CreditCardProcessor implementing PaymentProcessor
interface
class CreditCardProcessor(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing credit card payment of ${amount}")

    def refund_payment(self, amount):
        print(f"Refunding credit card payment of ${amount}")

# Concrete class PayPalProcessor implementing PaymentProcessor interface
class PayPalProcessor(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing PayPal payment of ${amount}")

    def refund_payment(self, amount):
        print(f"Refunding PayPal payment of ${amount}")

# Creating instances of different payment processors
cc_processor = CreditCardProcessor()
paypal_processor = PayPalProcessor()

# Using the interface methods
cc_processor.process_payment(100)  # Output: Processing credit card payment
of $100
cc_processor.refund_payment(50)    # Output: Refunding credit card payment
of $50
paypal_processor.process_payment(200)  # Output: Processing PayPal payment
of $200
paypal_processor.refund_payment(100)   # Output: Refunding PayPal payment
of $100
```

Explanation of Real-World Example:

- PaymentProcessor Interface:
    - Defines two abstract methods: process_payment() and refund_payment().

- CreditCardProcessor and PayPalProcessor Concrete Classes:
    - Inherit from PaymentProcessor and provide specific implementations for process_payment() and refund_payment(), representing behaviors specific to credit card and PayPal payment processing.
- Usage:
    - Creates instances cc_processor and paypal_processor of types CreditCardProcessor and PayPalProcessor.
    - Calls the process_payment() and refund_payment() methods on each instance, executing the implementation specific to each subclass.

Benefits of Using Interfaces:

1. Encapsulation: Interfaces encapsulate method signatures, hiding the details of their implementation.
2. Decoupling: By relying on interfaces rather than concrete implementations, code becomes more modular and easier to maintain or extend.
3. Polymorphism: Interfaces allow objects of different classes to be treated as instances of the same class through a common interface, facilitating polymorphic behavior.

In Python, although interfaces are not explicitly defined, the use of abstract classes and abstract methods provides a powerful way to enforce method contracts and ensure consistent interfaces across different classes.

10. EXCEPTION HANDLING

Exception handling allows developers to manage errors gracefully and ensure that a program can continue to run or fail gracefully. In Python, exceptions are events that can alter the flow of a program. They are typically used to handle errors and other exceptional conditions.

10.1. Understanding Exceptions

- An exception is an error that occurs during the execution of a program.
- When an error occurs, Python stops the program and generates an error message, known as an exception.
- Exceptions can be caused by many factors, such as incorrect input, file not found, division by zero, etc.

Why Handle Exceptions?

- To prevent the program from crashing and provide a user-friendly error message.
- To manage and recover from errors gracefully, allowing the program to continue running if possible.
- To ensure resources like file handles or network connections are properly released or closed.

Common Built-in Exceptions:

Python has several built-in exceptions. Here are a few commonly used ones:

- **ZeroDivisionError:** Raised when division by zero is attempted.
- **IndexError:** Raised when an index is out of range in a sequence.
- **KeyError:** Raised when a key is not found in a dictionary.
- **TypeError:** Raised when an operation or function is applied to an object of inappropriate type.

- **ValueError:** Raised when a function receives an argument of the right type but inappropriate value.
- **FileNotFoundError:** Raised when an attempt to open a file fails.

Examples of Common Exceptions:

1. ZeroDivisionError

```
print(10 / 0)  # This will raise ZeroDivisionError
```

Why?: Division by zero is mathematically undefined. When you try to divide a number by zero in Python, it raises a ZeroDivisionError.

2, IndexError:

```
my_list = [1, 2, 3]
print(my_list[5])  # This will raise IndexError
```

Why?: This error occurs because you are trying to access an index that does not exist in the list. my_list has only three elements (indexes 0, 1, and 2), so index 5 is out of range..

3. KeyError

```
my_dict = {"name": "Alice"}
print(my_dict["age"])  # This will raise KeyError
```

Why?: A KeyError occurs when you try to access a dictionary key that does not exist. In this case, the key "age" is not present in my_dict.

4. TypeError:

```
my_dict = {"name": "Alice"}
print(my_dict["age"])  # This will raise KeyError
```

Why?: A KeyError occurs when you try to access a dictionary key that does not exist. In this case, the key "age" is not present in my_dict.

5. ValueError:

```
number = int("abc")  # This will raise ValueError
```

Why?: A ValueError occurs when you pass an argument of the correct type but with an inappropriate value. Here, the int() function expects a string that represents a number, but "abc" is not a valid number.

6. FileNotFoundError:

```
with open("non_existent_file.txt", "r") as file:
    content = file.read()  # This will raise FileNotFoundError
```

Why?: This error occurs when you try to open a file that does not exist. Python raises a FileNotFoundError because it cannot find non_existent_file.txt.

10.2. Try, except, else, and finally blocks

Understanding how to use try, except, else, and finally blocks is crucial for handling exceptions and managing errors in Python effectively. These blocks allow you to catch and handle exceptions, ensuring that your code can respond appropriately to unexpected conditions.

1. try Block

- The try block lets you test a block of code for errors.
- If an error occurs, the code inside the try block stops executing, and the except block is executed.

2. except Block

- The except block lets you handle the error.
- You can specify different types of exceptions to handle specific errors.

3. else Block

- The else block lets you execute code if no errors were raised in the try block.
- It is executed only if the try block does not raise an exception.

4. finally Block

- The finally block lets you execute code, regardless of the result of the try and except blocks.
- It is typically used for cleanup actions that must be performed under all circumstances (e.g., closing a file).

Basic Syntax:

In Python, you handle exceptions using try, except, else, and finally blocks.

```
try:
```

```
    # Code that might raise an exception
    pass
except ExceptionType:
    # Code that runs if the exception occurs
    pass
else:
    # Code that runs if no exception occurs
    pass
finally:
    # Code that runs no matter what
    pass
```

Example:

```
try:
    number = int(input("Enter a number: "))  # This might raise a
ValueError
except ValueError:
    print("That's not a valid number!")  # Handle the ValueError
else:
    print(f"You entered {number}.")  # Run this if no exception occurs
finally:
    print("This will run no matter what.")  # Always execute this block
```

Output:

1. When the user enters a valid number:

```
Enter a number: 42
You entered 42.
This will run no matter what.
```

2. When the user enters an invalid number (e.g., letters or special characters):

```
Enter a number: abc
That's not a valid number!
This will run no matter what.
```

Explanation:

- try Block:
    - The try block contains code that might raise an exception. Here, it attempts to convert user input to an integer.
    - If the input is not a valid integer, a ValueError will be raised.
- except Block:
    - The except block catches and handles the exception specified. Here, it catches a ValueError.
    - If a ValueError is raised in the try block, this block will be executed, printing "That's not a valid number!".
- else Block:
    - The else block runs if no exception occurs in the try block.
    - If the input is valid, it will print the entered number.
- finally Block:
    - The finally block contains code that runs no matter what, whether an exception occurs or not.
    - Here, it prints "This will run no matter what."

Real-World Example:

Consider a scenario where we want to read data from a file. We need to handle potential errors like the file not existing or issues with reading the file.

```
def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            data = file.read()
            print(data)
    except FileNotFoundError:
        print("The file was not found.")
    except IOError:
        print("An I/O error occurred.")
    else:
        print("File read successfully.")
    finally:
        print("File reading operation is complete.")

# Using the function
```

```
read_file("example.txt")
```

Output:

1. When the file exists and is read successfully:

```
# Contents of example.txt: "Hello, world!"

Hello, world!
File read successfully.
File reading operation is complete.
```

2. When the file does not exist:

```
The file was not found.
File reading operation is complete.
```

3. When an I/O error occurs (e.g., permission issues):

```
An I/O error occurred.
File reading operation is complete.
```

Explanation:

- try Block:
    - Attempts to open and read the file specified by file_path.
    - If successful, it prints the contents of the file.
- except FileNotFoundError:
    - Catches and handles the FileNotFoundError if the file does not exist.
    - Prints a message indicating the file was not found.
- except IOError:
    - Catches and handles other I/O errors.
    - Prints a message indicating an I/O error occurred.
- else Block:

    - Executes if no exceptions are raised, indicating successful file reading.
    - Prints a success message.
- finally Block:
    - Executes regardless of whether an exception was raised or not.
    - Prints a message indicating the completion of the file reading operation.

Problem Statement

Write a program that prompts the user to input two numbers and then performs division. Handle the following exceptions:

1. ValueError if the input is not a number.
2. ZeroDivisionError if the second number is zero.
3. Ensure that the program always prints a message indicating that the operation is complete, regardless of whether an exception was raised or not.

Code:

```
def divide_numbers():
    try:
        # Prompt the user for the first number
        num1 = float(input("Enter the first number: "))

        # Prompt the user for the second number
        num2 = float(input("Enter the second number: "))

        # Perform the division
        result = num1 / num2

    except ValueError:
        # Handle the case where the input is not a number
        print("Invalid input! Please enter numeric values.")

    except ZeroDivisionError:
        # Handle the case where division by zero is attempted
        print("Error! Division by zero is not allowed.")
```

```
    else:
        # Print the result if no exceptions occur
        print(f"The result of the division is: {result}")

    finally:
        # Always print this message, regardless of whether an exception
occurred or not
        print("Operation complete.")

# Call the function to test it
divide_numbers()
```

Explanation:

- try Block:
  - The try block contains the code that could potentially raise an exception.
  - It prompts the user to input two numbers and attempts to convert them to float.
  - It then attempts to divide the first number by the second number.
- except Block for ValueError:
  - This block catches the ValueError exception.
  - If the user inputs a value that cannot be converted to float, a ValueError is raised, and this block prints an appropriate error message.
- except Block for ZeroDivisionError:
  - This block catches the ZeroDivisionError exception.
  - If the second number is zero, a ZeroDivisionError is raised during the division, and this block prints an appropriate error message.
- else Block:
  - The else block is executed only if no exceptions are raised in the try block.
  - It prints the result of the division.
- finally Block:
  - The finally block is executed regardless of whether an exception was raised or not.
  - It prints a message indicating that the operation is complete.

Outputs

1. Valid Input:

```
Enter the first number: 10
Enter the second number: 2
The result of the division is: 5.0
Operation complete.
```

2. Invalid Input (Non-numeric):

```
Enter the first number: ten
Invalid input! Please enter numeric values.
Operation complete.
```

3. Division by Zero:

```
Enter the first number: 10
Enter the second number: 0
Error! Division by zero is not allowed.
Operation complete.
```

10.3. Raising exceptions

Raising exceptions allows you to signal that an error or unexpected condition has occurred during the execution of your program. This can be useful for handling situations where certain conditions should not happen under normal circumstances. Let's dive into how raising exceptions works in Python.

Raising an exception means deliberately causing an exception to occur at a specific point in your code. This is typically done using the raise statement followed by an exception class or an instance of an exception.

Why Raise Exceptions?

1. Error Signaling: Raise exceptions to signal errors or exceptional conditions that need to be handled.
2. Control Flow: Exception handling allows you to control program flow and handle errors gracefully.
3. Debugging: Exceptions provide valuable information about what went wrong and where.

Syntax for Raising Exceptions

To raise an exception, you use the raise statement followed by an instance of the exception class. The general syntax is:

```
raise ExceptionType("Error message")
```

Example: Simple Use Case

Here's a simple example of raising an exception when a user tries to withdraw more money than available in their account.

```
class InsufficientFundsError(Exception):
    # Custom exception class for insufficient funds
    pass
```

```
def withdraw_money(balance, amount):
    if amount > balance:
        raise InsufficientFundsError("Insufficient funds in your account!")
    balance -= amount
    return balance

# Driver code
try:
    current_balance = 1000
    withdrawal_amount = 1500
    new_balance = withdraw_money(current_balance, withdrawal_amount)
    print(f"New balance: {new_balance}")
except InsufficientFundsError as e:
    print(e)  # Output: Insufficient funds in your account!
```

Explanation

- Custom Exception Class:
    - A custom exception class InsufficientFundsError is defined by inheriting from the base Exception class.
- Function withdraw_money:
    - This function takes the current balance and the amount to withdraw as parameters.
    - If the withdrawal amount exceeds the current balance, an InsufficientFundsError is raised with a specific error message.
    - If there are sufficient funds, the balance is reduced by the withdrawal amount, and the new balance is returned.
- Driver Code:
    - The driver code tries to withdraw money using the withdraw_money function.
    - If an InsufficientFundsError is raised, it is caught by the except block, and the error message is printed

Raising Custom Exceptions

You can write your custom exception to raise an error.

```
#Raising a custom exception
class InsufficientBalanceError(Exception):
    def __init__(self, message="Insufficient balance."):
        self.message = message
        super().__init__(self.message)

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientBalanceError()
    return balance - amount

# Test the function
try:
    current_balance = 1000
    amount_to_withdraw = 1500
    remaining_balance = withdraw(current_balance, amount_to_withdraw)
    print(f"Withdraw successful. Remaining balance: {remaining_balance}")
except InsufficientBalanceError as e:
    print(e)

# Output:
# Insufficient balance.
```

- **super()** in Python is used to access and call methods and attributes of a parent or superclass from within a subclass.
- It facilitates code reuse and maintains the inheritance hierarchy by allowing subclasses to extend or override methods defined in the superclass.
- Typically used inside methods of a subclass, super() helps in invoking methods from the superclass that have been overridden in the subclass.
- Supports Method Overriding: Allows subclasses to customize or extend the behavior of methods inherited from the superclass while retaining the original functionality.

.

Real-World Example: File Operations

Let's look at a real-world example where we raise exceptions in file operations to ensure that the file content is not empty.

```
class EmptyFileError(Exception):
```

```
    # Custom exception class for empty files
    pass

def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            data = file.read()
            if not data:
                raise EmptyFileError("The file is empty")
            print(data)
    except FileNotFoundError:
        print("The file was not found.")
    except EmptyFileError as e:
        print(e)  # Output: The file is empty (if the file is empty)
    except IOError:
        print("An I/O error occurred.")
    finally:
        print("File reading operation is complete.")

# Using the function
read_file("example.txt")
```

Explanation

- Custom Exception Class:
    - A custom exception class EmptyFileError is defined by inheriting from the base Exception class.
- Function read_file:
    - This function takes a file_path parameter and attempts to read the file.
    - If the file is found but empty, an EmptyFileError is raised with a specific error message.
    - If the file is not found, a FileNotFoundError is caught, and an error message is printed.
    - If any I/O error occurs, an IOError is caught, and an error message is printed.
    - The finally block prints a message indicating that the file reading operation is complete, regardless of whether an exception was raised or not.

11. LOGGING IN PYTHON

Logging is a crucial part of software development that helps developers track and record important events and actions within an application. It provides visibility into the runtime behavior of your code, aiding in debugging, auditing, and monitoring.

Why Use Logging?

- Debugging: Helps track down bugs and issues by providing insights into the flow of execution and the state of variables.
- Monitoring: Provides runtime information and metrics about the application's behavior.
- Auditing: Logs can serve as a record of actions and events for security and compliance purposes.
- Troubleshooting: Useful for investigating errors and exceptions encountered during application execution.

Components of Python Logging

- Loggers: Entry points in the logging system that you use to emit logs.
- Handlers: Directs where the log messages go (e.g., console, files, databases).
- Formatters: Specifies the layout and structure of log messages.
- Filters (optional): Provides a way to selectively suppress or pass log messages based on certain criteria.

Logging Levels

Python's logging module provides several predefined logging levels to categorize log messages based on their severity:

1. DEBUG: Detailed information, typically useful only for debugging.
2. INFO: Confirmation that things are working as expected.
3. WARNING: Indication that something unexpected happened, or indicative of a potential problem.
4. ERROR: Designates errors that caused the application to behave unexpectedly.
5. CRITICAL: Severe errors that may lead to the termination of the application.

There are two different ways in logging, they are:

1. Logging to console.
2. Logging to File.

Logging to Console

Logging to the console involves directing log messages to the standard output (console) during the execution of a Python program. It's useful for debugging and monitoring applications without writing log files.

1. Basic Logging Setup

```
import logging

# Configure logging to console
logging.basicConfig(level=logging.DEBUG)

# Example function with logging
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        logging.error("Division by zero occurred")
    else:
        logging.info(f"Division result: {result}")

# Test the function
divide(10, 2)
divide(10, 0)

#Output:
INFO:root:Division result: 5.0
ERROR:root:Division by zero occurred
```

Explanation:

- basicConfig(level=logging.DEBUG): Configures logging to display messages from the DEBUG level and above to the console.

- logging.error("Division by zero occurred"): Logs an error message when a ZeroDivisionError occurs during division.
- logging.info(f"Division result: {result}"): Logs an info message with the division result when no error occurs.

2. Logging with Formatting

```
import logging

# Configure logging with format
logging.basicConfig(format='%(asctime)s - %(levelname)s - %(message)s',
level=logging.INFO)

# Example function with logging
def process_data(data):
    try:
        # Process data
        result = data['key']
    except KeyError as e:
        logging.error(f"KeyError: {e}")
    else:
        logging.info("Data processed successfully")

# Test the function
process_data({'key': 'value'})
process_data({'wrong_key': 'value'})

#Output:
2024-07-16 12:00:00,000 - ERROR - KeyError: 'key'
2024-07-16 12:00:01,000 - INFO - Data processed successfully
```

Explanation:

- basicConfig(format='%(asctime)s - %(levelname)s - %(message)s', level=logging.INFO): Configures logging with a custom format (asctime, levelname, message) and sets the logging level to INFO.
- logging.error(f"KeyError: {e}"): Logs an error message with details when a KeyError occurs while processing data.
- logging.info("Data processed successfully"): Logs an info message when data is processed successfully.

3. Integrating with Exception Handling

```
import logging

# Configure logging with format and level
logging.basicConfig(format='%(asctime)s - %(levelname)s - %(message)s',
level=logging.DEBUG)

# Example function with exception handling and logging
def fetch_data(data_list, index):
    try:
        result = data_list[index]
    except IndexError as e:
        logging.error(f"IndexError: {e}")
    else:
        logging.info(f"Data fetched: {result}")

# Test the function
data = [1, 2, 3]
fetch_data(data, 1)
fetch_data(data, 5)

# Output:
2024-07-16 12:00:00,000 - INFO - Data fetched: 2
2024-07-16 12:00:01,000 - ERROR - IndexError: list index out of range
```

Explanation:

- logging.basicConfig(format='%(asctime)s - %(levelname)s - %(message)s', level=logging.DEBUG): Configures logging with a timestamp, log level, and message format, setting the logging level to DEBUG.
- logging.error(f"IndexError: {e}"): Logs an error message with details when an IndexError occurs while fetching data from the list.
- logging.info(f"Data fetched: {result}"): Logs an info message with the fetched data when no error occurs.

4. Logging with Try-Except-Finally

```
import logging
```

```
# Configure logging with basic settings
logging.basicConfig(level=logging.INFO)

# Example function with logging in try-except-finally
def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            data = file.read()
            logging.info(f"File read successfully: {file_path}")
            return data
    except FileNotFoundError:
        logging.error(f"File not found: {file_path}")
    except IOError as e:
        logging.error(f"I/O error occurred: {e}")
    finally:
        logging.info("File reading operation completed.")

# Test the function
read_file("example.txt")
read_file("non_existent_file.txt")

# Output:
INFO:root:File read successfully: example.txt
INFO:root:File reading operation completed.
ERROR:root:File not found: non_existent_file.txt
INFO:root:File reading operation completed.
```

Explanation:

- logging.basicConfig(level=logging.INFO): Configures logging to display INFO level messages and above.
- logging.info(f"File read successfully: {file_path}"): Logs an info message when a file is successfully read.
- logging.error(f"File not found: {file_path}"): Logs an error message when a FileNotFoundError occurs.
- logging.error(f"I/O error occurred: {e}"): Logs an error message with details when an I/O error occurs during file reading.
- logging.info("File reading operation completed."): Logs an info message indicating completion of file reading operation.

Logging to file

Logging to a file involves directing log messages to a specified file instead of the console. This is useful for persistently storing logs for analysis, debugging, and auditing purposes.

Basic Logging to File Setup

```
import logging

# Configure logging to a file
logging.basicConfig(filename='app.log', level=logging.DEBUG)

# Example function with logging
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        logging.error("Division by zero occurred")
    else:
        logging.info(f"Division result: {result}")

# Test the function
divide(10, 2)
divide(10, 0)

# Output:
INFO:root:Division result: 5.0
ERROR:root:Division by zero occurred
```

Explanation:

- basicConfig(filename='app.log', level=logging.DEBUG): Configures logging to write messages to a file named app.log with DEBUG level and above.
- logging.error("Division by zero occurred"): Logs an error message when a ZeroDivisionError occurs during division.
- logging.info(f"Division result: {result}"): Logs an info message with the division result when no error occurs.

2. Logging with Formatting to File

```
import logging

# Configure logging to file with format
logging.basicConfig(filename='app.log', format='%(asctime)s - %(levelname)s
- %(message)s', level=logging.INFO)

# Example function with logging
def process_data(data):
    try:
        # Process data
        result = data['key']
    except KeyError as e:
        logging.error(f"KeyError: {e}")
    else:
        logging.info("Data processed successfully")

# Test the function
process_data({'key': 'value'})
process_data({'wrong_key': 'value'})

#Output:
2024-07-16 12:00:00,000 - ERROR - KeyError: 'key'
2024-07-16 12:00:01,000 - INFO - Data processed successfully
```

Explanation:

- basicConfig(filename='app.log', format='%(asctime)s - %(levelname)s - %(message)s', level=logging.INFO): Configures logging with a timestamp, log level, and message format, writing to app.log file with INFO level and above.
- logging.error(f"KeyError: {e}"): Logs an error message with details when a KeyError occurs while processing data.
- logging.info("Data processed successfully"): Logs an info message when data is processed successfully.

3. Integrating with Exception Handling and Logging to File

```
import logging

# Configure logging to file with format and level
logging.basicConfig(filename='app.log', format='%(asctime)s - %(levelname)s
```

```
- %(message)s', level=logging.DEBUG)

# Example function with exception handling and logging
def fetch_data(data_list, index):
    try:
        result = data_list[index]
    except IndexError as e:
        logging.error(f"IndexError: {e}")
    else:
        logging.info(f"Data fetched: {result}")

# Test the function
data = [1, 2, 3]
fetch_data(data, 1)
fetch_data(data, 5)

# Output:
2024-07-16 12:00:00,000 - INFO - Data fetched: 2
2024-07-16 12:00:01,000 - ERROR - IndexError: list index out of range
```

Explanation:

- logging.basicConfig(filename='app.log', format='%(asctime)s - %(levelname)s - %(message)s', level=logging.DEBUG): Configures logging with a timestamp, log level, and message format, writing to app.log file with DEBUG level and above.
- logging.error(f"IndexError: {e}"): Logs an error message with details when an IndexError occurs while fetching data from the list.
- logging.info(f"Data fetched: {result}"): Logs an info message with the fetched data when no error occurs.

4. Logging with Try-Except-Finally to File

```
import logging

# Configure logging to file with basic settings
logging.basicConfig(filename='app.log', level=logging.INFO)

# Example function with logging in try-except-finally
def read_file(file_path):
    try:
```

```
        with open(file_path, 'r') as file:
            data = file.read()
            logging.info(f"File read successfully: {file_path}")
            return data
    except FileNotFoundError:
        logging.error(f"File not found: {file_path}")
    except IOError as e:
        logging.error(f"I/O error occurred: {e}")
    finally:
        logging.info("File reading operation completed.")

# Test the function
read_file("example.txt")
read_file("non_existent_file.txt")

# Output:
INFO:root:File read successfully: example.txt
INFO:root:File reading operation completed.
ERROR:root:File not found: non_existent_file.txt
INFO:root:File reading operation completed.
```

Explanation:

- logging.basicConfig(filename='app.log', level=logging.INFO): Configures logging to write INFO level messages and above to app.log file.
- logging.info(f"File read successfully: {file_path}"): Logs an info message when a file is successfully read.
- logging.error(f"File not found: {file_path}"): Logs an error message when a FileNotFoundError occurs.
- logging.error(f"I/O error occurred: {e}"): Logs an error message with details when an I/O error occurs during file reading.
- logging.info("File reading operation completed."): Logs an info message indicating completion of file reading operation.

12. REGULAR EXPRESSIONS

12.1. Pattern matching with regular expressions

Regular expressions are sequences of characters that define a search pattern. They are powerful tools for string manipulation and allow you to match patterns within text efficiently. Here's an in-depth look at how they work:

Definitios:

- Regex: A sequence of characters that define a search pattern.
- Literal Characters: Characters that match themselves exactly (e.g., letters, digits, symbols).
- Metacharacters: Special characters with unique meanings in regex (e.g., . for any character, ^ for start of a line, $ for end of a line).

Common Metacharacters:

Metacharacters are special characters in regular expressions that have a specific meaning or function.

- . (Dot): Matches any single character except newline.
- ^: Matches the start of the string.
- $: Matches the end of the string.
- \d: Matches any digit (0-9).
- \w: Matches any alphanumeric character (word characters).
- \s: Matches any whitespace character (space, tab, newline).

Quantifiers:

Quantifiers specify the number of occurrences of a character or group in a regex pattern.

- *: Matches zero or more occurrences of the preceding element.
- +: Matches one or more occurrences of the preceding element.
- ?: Matches zero or one occurrence of the preceding element.
- {m}: Matches exactly m occurrences of the preceding element.
- {m,}: Matches m or more occurrences of the preceding element.

- {m,n}: Matches between m and n occurrences of the preceding element.

Character Classes:

A character class in regex defines a set of characters to match.

- [...]: Matches any single character inside the brackets.
- [^...]: Matches any single character not inside the brackets.
- \d: Matches any digit ([0-9]).
- \w: Matches any alphanumeric character and underscore ([a-zA-Z0-9_]).
- \s: Matches any whitespace character ([\t\n\r\f\v]).

Anchors and Boundaries:

Anchors and boundaries in regex define specific positions in a string where matches should occur.

- ^: Matches the start of the string.
- $: Matches the end of the string.
- \b: Matches a word boundary (the position between a word character and a non-word character).

Using the re Module in Python:

The re module in Python provides functions and methods for working with regular expressions.

- re.search(pattern, string): Searches for a pattern within a string.
- re.match(pattern, string): Matches a pattern only at the beginning of a string.
- re.findall(pattern, string): Finds all occurrences of a pattern in a string.
- re.finditer(pattern, string): Finds all occurrences of a pattern and returns match objects.
- re.sub(pattern, repl, string): Replaces occurrences of a pattern in a string with another substring.

Flags and Options:

Flags and options modify how the regex engine interprets the pattern matching.

- re.IGNORECASE: Perform case-insensitive matching.
- re.MULTILINE: Allow ^ and $ to match the start and end of each line.
- re.DOTALL: Allow . to match newline characters as well.

Applications:

- Validation: Verify if a string matches a specific format (e.g., email addresses, phone numbers).
- Extraction: Extract portions of text that match certain criteria (e.g., extracting URLs from a webpage).
- Substitution: Replace parts of a string based on a pattern (e.g., replacing all occurrences of a word).
- Splitting: Divide a string into substrings based on a pattern (e.g., splitting a sentence into words).

12.2. Using the ‘re’ Module

Python's ‘re’ module provides robust support for working with regular expressions. It allows you to perform various operations such as searching, matching, finding all occurrences, and substitution within strings using regex patterns. Here's a detailed exploration of using the re module:

1. Importing the Module:

- To use the re module, you need to import it first:

```
import re
```

2. Key Functions:

- re.search(pattern, string, flags=0):
    - Searches for the first occurrence of pattern in string.
    - Returns a match object if found, None otherwise.

```
result = re.search(r'is', 'This is a test.')
print(result.group())  # Output: is
```

- re.match(pattern, string, flags=0):
    - Matches pattern at the beginning of string.
    - Returns a match object if found, None otherwise.

```
result = re.match(r'This', 'This is a test.')
print(result.group())  # Output: This
```

- re.findall(pattern, string, flags=0):
    - Finds all occurrences of pattern in string.
    - Returns a list of all matches.

```
result = re.findall(r'\d+', 'There are 5 apples and 12 oranges.')
print(result)  # Output: ['5', '12']
```

- re.finditer(pattern, string, flags=0):
    - Returns an iterator yielding match objects for all matches of pattern in string.

```
iterator = re.finditer(r'\d+', 'There are 5 apples and 12 oranges.')
for match in iterator:
    print(match.group())  # Output: 5, 12
```

- re.sub(pattern, repl, string, count=0, flags=0):
    - Replaces occurrences of pattern in string with repl.
    - Returns the modified string.

```
result = re.sub(r'\d+', 'X', 'There are 5 apples and 12 oranges.')
print(result)  # Output: There are X apples and X oranges.
```

3. Regex Objects:

- re.compile(pattern, flags=0):
    - Compiles a regex pattern into a regex object.
    - Allows for reuse of the compiled regex pattern across multiple operations, which can improve performance for repeated operations.

```
pattern = re.compile(r'\d+')
result = pattern.findall('There are 5 apples and 12 oranges.')
print(result)  # Output: ['5', '12']
```

4. Flags and Options:

- Flags modify the behavior of regex patterns:
    - re.IGNORECASE: Ignores case sensitivity during matching.
    - re.MULTILINE: Treats the string as multiple lines (^ and $ match the start/end of each line).
    - re.DOTALL: Allows . to match newline characters (\n).

```
result = re.findall(r'apple', 'An apple a day keeps the doctor away.',
flags=re.IGNORECASE)
```

```
print(result)  # Output: ['apple']
```

5. Error Handling:

- Use error handling to manage cases where regex operations might fail due to invalid patterns or unexpected input.

```
try:
    result = re.findall('(', 'Invalid regex pattern (')
except re.error as e:
    print(f"Regex error: {e}")
```

Example:

```
import re

# String containing alphanumeric characters
text = "Hello 123, this is a test string with numbers 456 and some
punctuation marks!"

# Using re.findall to find all sequences of digits (\d+)
digits = re.findall(r'\d+', text)

# Printing the found digits
print("Digits found in the text:", digits)
```

Output:

```
Digits found in the text: ['123', '456']
```

Real-World Example

Finding Email Addresses in Text

```
import re

# Sample text containing email addresses
text = """
Contact us at support@example.com for any inquiries.
For business opportunities, email business@company.com.
```

```
You can also reach us at info@organization.org.
"""

# Using re.findall to find all email addresses
(\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b)
emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b',
text)

# Printing the found email addresses
print("Email addresses found in the text:")
for email in emails:
    print(email)
```

Explanation:

- Importing the re Module:
  - import re: Imports the regular expression module.
- Defining the Sample Text:
  - text = """ ... """: Defines a multi-line string containing sample text with email addresses.
- Using re.findall:
  - emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text): Uses re.findall to find all email addresses in the text string.
  - The regex pattern \b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b matches email addresses based on common email format rules:
  - \b: Word boundary ensures the entire email address is matched.
  - [A-Za-z0-9._%+-]+: Matches one or more alphanumeric characters, dots, underscores, percent signs, plus signs, or hyphens (local part of the email).
  - @: Matches the "@" symbol.
  - [A-Za-z0-9.-]+: Matches one or more alphanumeric characters, dots, or hyphens (domain part of the email).
  - \.: Matches a literal dot (before the domain extension).
  - [A-Z|a-z]{2,}: Matches the top-level domain (TLD) with at least 2 characters (e.g., .com, .org, .net).
- Printing the Result:

- print("Email addresses found in the text:"): Prints a header indicating the following email addresses.
- for email in emails:: Iterates over the list of found email addresses.
- print(email): Prints each email address found in the text.

Output:

```
Email addresses found in the text:
support@example.com
business@company.com
info@organization.org
```

Problem Statement:

Write a Python program that reads a text input and prints all words that start with the letter 'S'. Assume the input text can contain multiple sentences.

Code:

```
import re

def find_words_starting_with_s(text):
    # Using regular expression to find words starting with 'S' or 's'
    words_starting_with_s = re.findall(r'\b[Ss]\w+', text)

    # Printing the found words
    print("Words starting with 'S' found in the text:")
    for word in words_starting_with_s:
        print(word)

# Sample text input
text = """
Sharks are fascinating creatures, especially the Great White shark.
She sells seashells by the seashore. Sally likes swimming in the sea.
"""

# Finding and printing words starting with 'S'
find_words_starting_with_s(text)
```

Output:

```
Words starting with 'S' found in the text:
Sharks
shark
She
sells
seashells
seashore
Sally
swimming
sea
```

- Using ‘re.findall’:
    - words_starting_with_s = re.findall(r'\b[Ss]\w+', text): Uses re.findall with a regex pattern \b[Ss]\w+ to find all words that start with 'S' or 's'.
    - \b: Word boundary ensures the entire word is matched.
    - [Ss]: Matches either 'S' or 's'.
    - \w+: Matches one or more alphanumeric characters (letters, digits, or underscores) following the initial 'S' or 's'.