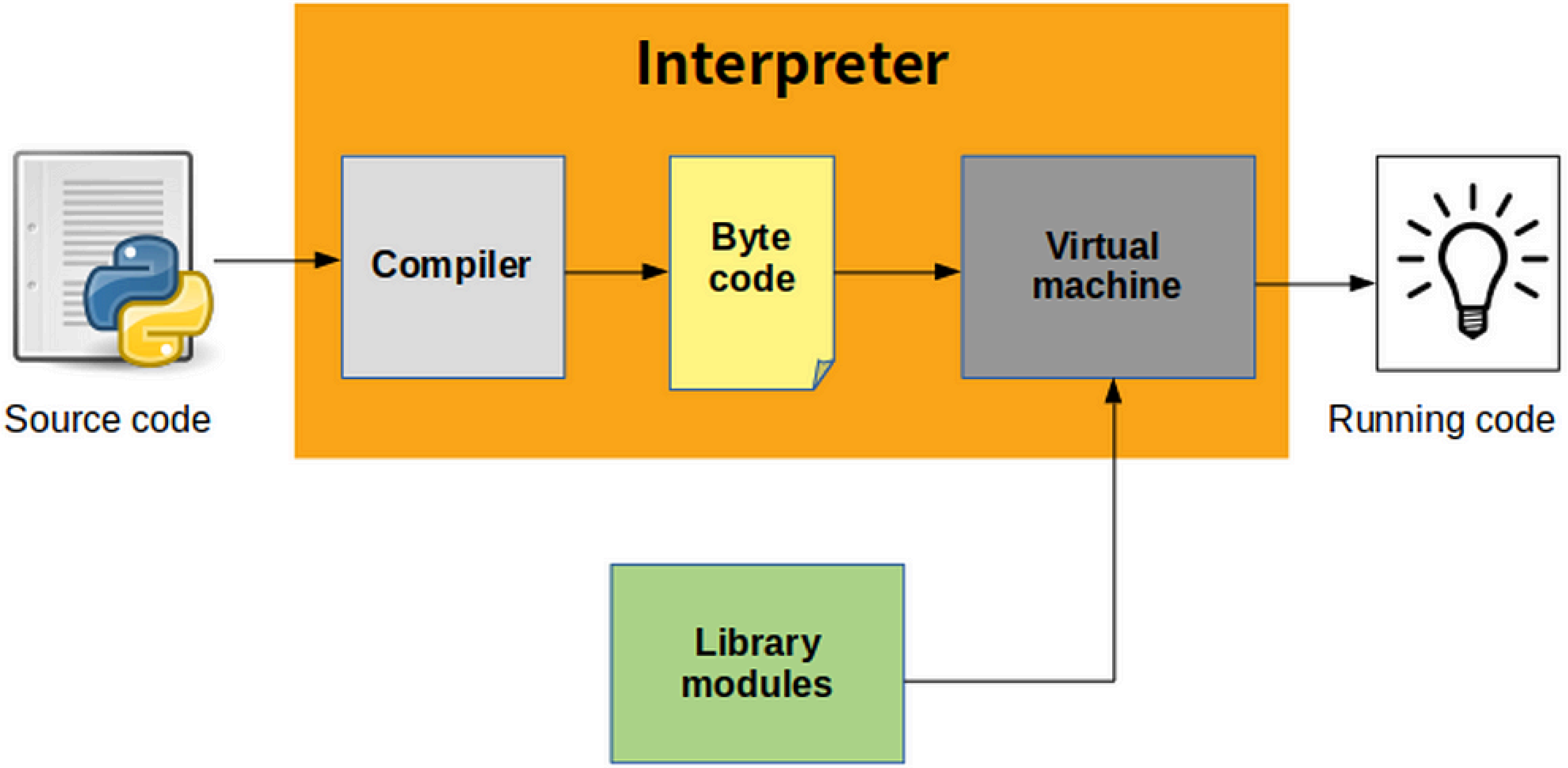
What is Python?

- Python is simple & easy
- High Level Programming, Object Oriented & Scripting Language
- Free & Open Source
- Developed by Guido van Rossum in 1991. Name is Taken from Monty Python Circus Show in BBC.
- Portable & Versatile

Python Virtual Machine



Why Python?

Python is a popular programming language known for its simplicity, readability, and versatility. Its numerous benefits and features have gained widespread adoption in various fields. Some of the main reasons why Python is widely used are:

1. Web Development
2. Game Development
3. Artificial Intelligence and Machine Learning

Day - 2

Where should I run Python Code?

- VS Code is Preferable.

Install Python & VS Code on PC

Run Our First Program

```
print("This is my first program in Python")
```

Output:

```
This is my first program in Python
```

Common Mistakes

- Spelling Mistakes: prnt(”Hello World!”)
- Uppercase ‘P’: Print(”Hello World!”)
- Missing quotes: print(Hello World!)

Calculations

We can use Python for doing addition, subtraction, multiplication, divisions also

```
print(2+5)
#Output: 7
print(2-5)
#Output: -3
print(2*5)
#Output: 10
print(6/3)
#Output: 2.0
```

Assigning Variables

We can use Python for assigining multiple variables also

```
a = "Nihar"
b = "Loves"
c = "Python"
print(a+" "+b+" "+c)
#Output: Nihar Loves Python
```

Python Character Set

- Letters – A to Z, a to z
- Digits – 0 to 9
- Special Symbols - + - * / etc.
- Whitespaces – Blank Space, tab, carriage return, newline, formfeed
- Other characters – Python can process all ASCII and Unicode characters as part of data or literals

Questions to Practice :

- Print your name.
- Print the result of adding two numbers.
- Print the result of subtracting two numbers.
- Print the result of multiplying two numbers.
- Print the result of dividing two numbers.

Day - 3

Variables

A variable is a name given to a memory location in a program.

```
name = "Siva"
age = 23
price = 25.99
```

Memory

```
name = "Siva"
age = 38
price = 25.99
```

Data Types

Mainly there are 4 types of Data Types:

1. **Strings :** Stream of Characters

   eg: “Siva”, “My name is Siva”

- Capital Letters (A – Z)
- Small Letters (a – z)
- Digits (0 – 9)
- Special Characters (~ ! @ # $ % ^ . ?,)
- Space

2. **Integer** (Example ki 1,2,-9,0)
3. **Float** (Any number with Decimal points, Example ki -3.90, 4.5)
4. **Boolean** (True, False)

```
# Integer data type
my_integer = 42
print(my_integer)  # Output: 42

# Float data type
my_float = 3.14
print(my_float)  # Output: 3.14

# String data type
my_string = "Hello, World!"
print(my_string)  # Output: Hello, World!

# Boolean data type
is_true = True
is_false = False
print(is_true)   # Output: True
print(is_false)  # Output: False
```

Questions to Practice :

1. Declare two variables a and b, assign integer values to them, and print their sum.

- Expected Output: The sum of a and b.

Code:
```
# Declare variables and assign integer values
a = 5
b = 10

# Calculate and print their sum
print("The sum of a and b is:", a + b)
```

2. Create a variable name and assign your name to it. Print a greeting message using your name.

- Expected Output: Greeting message with your name, e.g., "Hello, John!"

Code:

```
# Assign your name to the variable
name = "John"

# Print a greeting message
print(f"Hello, {name}!")
```

3. Define a variable pi and assign the value of π (pi) to it. Print the value of pi.

- Expected Output: The value of π (pi), e.g., 3.14159.

Code:

```
# Import the math module to use the value of π (pi)
import math

# Assign the value of π to the variable
pi = math.pi

# Print the value of π
print(f"The value of pi is: pi")
```

4. Define a variable is_raining and ask the user to input either "True" or "False" (as a string). Convert the input to a boolean and print its type.

- Expected Input: "True" or "False"
- Expected Output: The data type of the converted boolean.

Code:

```
# Define the variable
is_raining = input("Is it raining? Enter 'True' or 'False': ")

# Convert the input string to a boolean
is_raining_boolean = is_raining == "True"

# Print the data type of the converted variable
print(f"The type of the variable is: {type(is_raining_boolean)}")
```

Day - 4

Rules for Identifiers

In Python, **identifiers** are the names used to identify variables, functions, classes, or other objects. Here are the rules and conventions for creating identifiers:

1. Allowed Characters

- Identifiers can only contain letters(A-Z, a-z), digits (0-9), and underscores (`_`).
- They cannot include special characters like `@`, `$`, `%`, or spaces.

2. Cannot Start with a Digit

- Identifiers must begin with a letter or an underscore.
- Example:
  - Valid: `name`, `_value`
  - Invalid: `1value`, `9name`

3. Case Sensitivity

- Identifiers are case-sensitive.

For example, `name`, `Name`, and `NAME` are treated as three different identifiers.

4. Reserved Words Are Not Allowed

- Python keywords (e.g., `if`, `else`, `class`, `def`, `for`) cannot be used as identifiers.
  - Example:
  - Invalid: `if = 10`
  - Valid: `if_value = 10`

5. No Length Limit

- Python does not impose a limit on the length of an identifier, but it's good practice to keep them concise yet descriptive.

6. Special Characters

- An identifier starting with a single underscore _ is treated as a "protected" identifier (by convention, used for internal purposes).
- An identifier starting with double underscores (__) is considered "private" by convention.
- If an identifier ends with double underscores (__), it usually represents a special method (e.g., __init__, __str__).

7. Best Practices (PEP 8)

- Use meaningful and descriptive names: e.g., total_sum, user_age.
- Use snake_case for variable and function names: e.g., my_variable, calculate_sum.
- Use CamelCase for class names: e.g., MyClass, StudentDetails.
- Avoid single-letter identifiers except for loop variables: e.g., i, j.

Examples of Valid and Invalid Identifiers:

Valid Identifiers	Invalid Identifiers
my_var	my-var (contains -)
value2	2value (starts with digit)
_privateVar	total% (contains %)
getTotal	get total (contains space)
ClassName	for (reserved keyword)

Following these rules ensures your code is syntactically correct and readable.

Order Of Operations

Python follows the order of operations PEMDAS or BODMAS :

- Parentheses: None in this expression.
- Exponents: None in this expression.
- Multiplication/Division(from left to right):

  3 * 2 = 6

  8 / 4 = 2
- Addition/Subtraction(from left to right):

  10 + 6 = 16

  16 - 2 = 14

- B - Brackets first
- O - Orders (exponents and roots, like square roots) next
- DM - Division and Multiplication, from left to right
- AS - Addition and Subtraction, from left to right

```
# Example expression: 10 + 5 * (2 ** 3) - 6 / 2

result = 10 + 5 * (2 ** 3) - 6 / 2

 Step 1: Evaluate the expression within the brackets first
       2 ** 3 = 8
      So, the expression becomes: 10 + 5 * 8 - 6 / 2

Step 2: Perform Multiplication
      5 * 8 = 40
      So, the expression becomes: 10 + 40 - 6 / 2

Step 3: Perform division
    6 / 2 = 3
    So, the expression becomes: 10 + 40 - 3

Step 4: Perform addition
    10 + 40 = 50
    So, the final result is: 50 - 3 = 47

print(result)  # Output: 47.0
```

Questions to Practice :

1. result = 10 + 5 * 2 - 12 / 4
2. result = 4 ** 3 + 5 / 4 * 3
3. result = (8 + 8) * 3 / 2
4. result = 16 / 4 + 2 ** 3 - 6
5. result = 10 - 3 * (4 + 2) / 5

Day - 5

String Concatenation

String concatenation refers to combining two or more strings into a single string. In Python, there are several ways to achieve string concatenation, each suitable for different use cases.

1. Using the + Operator

Description: The + operator is the simplest and most intuitive way to concatenate strings.

Example:

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)
```

Output:

```
Hello World
```

Limitations: This method can be inefficient for a large number of concatenations because strings are immutable in Python, leading to the creation of new strings each time.

2. Using the join() Method

Description: The join() method is often more efficient for concatenating multiple strings, especially when working with lists.

Example:

```
words = ["Python", "is", "awesome"]
result = " ".join(words)
print(result)
```

Output:

```
Python is awesome
```

Advantages: It is efficient and recommended for joining large numbers of strings.

3. Using f-strings (Formatted String Literals)

Description: Introduced in Python 3.6, f-strings allow embedding expressions inside string literals.

Example:

```
name = "Alice"
greeting = f"Hello, {name}!"
print(greeting)
```

Output:

```
Hello, Alice!
```

Advantages: Clean and readable, especially when combining strings with variables.

4. Using the % Operator (Old Style)

Description: The % operator is an older method for string formatting and concatenation

Example:

```
name = "Bob"
age = 25
result = "My name is %s and I am %d years old." % (name, age)
print(result)
```

Output:

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

Limitations: Less readable and has largely been replaced by str.format() and f-strings.

5. Using the str.format() Method

Description: A modern and versatile way to concatenate strings with variables.

Example:

```
name = "Charlie"
age = 30
result = "My name is {} and I am {} years old.".format(name, age)
print(result)
```

Output:

```
My name is Charlie and I am 30 years old.
```

Advantages: Supports advanced formatting and is widely used.

6. Using += for Incremental Concatenation

Description: You can use the += operator to append strings incrementally.

Example:

```
message = "Hello"
message += ", "
message += "World!"
print(message)
```

Output:

```
Hello, World!
```

Caution: Similar to the + operator, it can be inefficient for many operations due to the immutability of strings.

7. Using * for Repeating Strings

Description: Though not exactly concatenation, you can repeat strings using the * operator

Example:

```
word = "ha"
result = word * 3
print(result)
```

Output:

```
hahaha
```

Performance Considerations

Small Strings: The + operator and += are fine for a small number of strings.
Large Strings or Multiple Joins: Use join() for better performance.
Dynamic Concatenation: Use f-strings or str.format() for readability and convenience.

Questions to Practice :

1. Combine the string "I am " with the number 25 (as a string) to produce **"I am 25"**

2. **Create a variable city with the value "Paris" and concatenate it into the sentence: "I live in Paris."**

3.Combine "Python" with "!" repeated five times to create **"Python!!!!!"**.

4. Create a sentence using concatenation that says: **"Hello, my name is Sam. I am 25 years old and I love Python programming!**

5.Define two variables, animal = "cat" and sound = "meow". Combine them to form **"A cat says meow.".**

6. Define a variable first_name as "John" and last_name as "Doe". Concatenate them with a space to create **"John Doe".**

Day - 6

Input and Output in Python

Python provides simple and effective ways to interact with users through input and output functions.

Input

1. **input() Function:**

- Used to get input from the user.
- Always returns the input as a string.
- **Syntax :** variable = input("Prompt message: ")

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

2. Converting Input:

Since input() returns a string, you may need to convert it into the desired data type.

eg:
```
age = int(input("Enter your age: "))  # Converts input to an integer
height = float(input("Enter your height in meters: "))  # Converts input to a float.
```

3. Handling Multiple Inputs:

- Use split() to take multiple inputs in a single line.

eg:
```
a, b = input("Enter two numbers: ").split()
print("First number:", a)
print("Second number:", b)
```

Output

1. **print() Function:**

Used to display output on the screen.

eg: print("Welcome to Python Class!")

2. **Features of print():**

- sep: Specifies the separator between values. Default is a space.

```
print("Hello", "World", sep=", ")
# Output: Hello, World
```

- end: Specifies what to print at the end. Default is a newline (\n)

```
print("Hello", end="!")
print("World")
# Output: Hello!World
```

3. String Formatting: -Refer to Day - 5 Class-

Concatenation:

```
name = "Alice"
print("Hello, " + name)
```

f-strings (recommended):

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

format() Method:

```
city = "Paris"
print("I live in {}".format(city))
```

4. Printing Multiple Values:

eg:

```
a = 10
b = 20
print("The sum of", a, "and", b, "is", a + b)
# Output: The sum of 10 and 20 is 30
```

Input and Output Together

Eg:

```
name = input("What is your name? ")
age = int(input("How old are you? "))
print(f"Hello, {name}! You are {age} years old.")
```

Key Points:

- Input: Always a string, so type conversion may be necessary.
- Output: Flexible with string formatting (+, f-strings, format()).
- input() and print() are the most commonly used methods for interaction in basic Python programs.

Questions to Practice:

Basic Input:

- Ask the user to input their favorite hobby and print a confirmation message: "Your favorite hobby is [hobby]."

Input Conversion:

- Ask the user to input a number and print its square.

Multiple Inputs:

- Ask the user to input their first and last name in one line (separated by a space). Print the full name.

String Manipulation:

- Ask the user to enter a word and print the word in uppercase, lowercase, and reversed order.

Basic Output:

- Print a formatted message: "Welcome to Python programming!"

Using sep and end:

- Print the following in one line: "Python", "is", "awesome" using sep as " - ".

Formatted Output:

- Create variables item = "Laptop" and price = 75000. Print:
- "The price of a Laptop is ₹75000."

Dynamic Formatting:

- Use variables city = "Paris" and hobby = "painting" to print:
- "I live in Paris, and my hobby is painting."

Test - 1

1. Declare two variables, x and y, assign the values 10 and 5 to them, and print their sum.

Expected Output: 15

2. Assign your name to a variable called name and print:

"My name is [name]."

3. Swap the values of two variables a and b without using a third variable.

4. Create a variable country and assign your favorite country to it. Use this variable to print:

"I would love to visit [country] someday

5. What is the data type of the following values?

- 42
- 3.14
- "Hello, World!"
- True

6.Create a variable num and assign an integer to it. Convert it to a float and print the result.

7. Write a Python program to take the user's height (in meters) as input and print its type. Example input: 1.75.
8. Create a variable is_happy with a boolean value. Print the message:
"You are happy!" if is_happy is True, otherwise print: "You are not happy."
9. Which of the following are valid Python identifiers?
    - **my_variable**
    - **2cool4school**
    - **def**
    - **price$**
    - **_value**
10. Create a variable name following the rules of identifiers and assign it a value of your choice.
11. Explain why variable names like class and if are not valid in Python.
12. Rename this variable 1st_number to make it a valid Python identifier.

13. Write a program to ask the user for their favorite book and print:
"Your favorite book is [book]."

14. Take two numbers as input from the user and calculate their average.

15. Ask the user for their birth year and calculate their current age.
Hint: Subtract their birth year from the current year.

16. Write a program to take a user's name and age as input and print:
"Hello, [name]! You are [age] years old."

17. Create a variable temp_celsius and assign it a value. Convert it to Fahrenheit and print the result using the formula:

    F=9/5*C+32

18. Define a variable sentence containing any sentence of your choice. Ask the user to input a number, and print the sentence repeated that number of times.

19.Write a program to calculate the area of a triangle.

   - **Formula: Area=Base×Height/2**
   - Take Base and Height as input from the user.

Answers:

```
1 . x = 10
    y = 5
    print(x + y)  # Output: 15
2. name = "John"
    print(f"My name is {name}.")
3. a = 5
    b = 10
    a, b = b, a
    print(a, b)  # Output: 10 5
4. country = "Japan"
    print(f"I would love to visit {country} someday!")
5. 42: int
    3.14: float
    "Hello, World!": str
     True: bool
```

```
6. num = 42
    print(float(num))  # Output: 42.0
7. height = float(input("Enter your height in meters: "))
    print(f"The type of height is: {type(height)}")  # Output: <class 'float'>
8. is_happy = True
   if is_happy:
   print("You are happy!")
   else:
   print("You are not happy.")
9. my_variable: Valid
    2cool4school: Invalid (Cannot start with a number)
    def: Invalid (Reserved keyword)
    price$: Invalid (Special characters not allowed)
    _value: Valid
```

```
10. my_age = 25
       print(my_age)
```

11. Reserved keywords: Variables like class and if are reserved Python keywords. They cannot be used as variable names.

```
12. first_number = 10
       print(first_number)
13. book = input("What is your favorite book? ")
      print(f"Your favorite book is {book}.")
14. num1 = float(input("Enter the first number: "))
      num2 = float(input("Enter the second number: "))
      average = (num1 + num2) / 2
       print(f"The average is {average}.")
15. birth_year = int(input("Enter your birth year: "))
       current_year = 2024  # Change this to the current year
       age = current_year - birth_year
       print(f"You are {age} years old.")
```

```
16. name = input("Enter your name: ")
    age = int(input("Enter your age: "))
    print(f"Hello, {name}! You are {age} years old.")
17. temp_celsius = float(input("Enter temperature in Celsius: "))
    temp_fahrenheit = (9 / 5) * temp_celsius + 32
    print(f"The temperature in Fahrenheit is {temp_fahrenheit:.2f}.")
18. sentence = input("Enter a sentence: ")
    times = int(input("Enter the number of repetitions: "))
    print(sentence * times)
19. base = float(input("Enter the base of the triangle: "))
    height = float(input("Enter the height of the triangle: "))
    area = (base * height) / 2
     print(f"The area of the triangle is {area:.2f}.")
```

Day - 8

Real Time Applications

Program: Monthly Budget Calculator

This program calculates your monthly savings based on your income and expenses.

```
# Input: User's income and expenses
   print("Welcome to the Monthly Budget Calculator!")
   monthly_income = float(input("Enter your monthly income (in $): "))
   rent_expense = float(input("Enter your rent expense (in $): "))
   food_expense = float(input("Enter your food expense (in $): "))
   transport_expense = float(input("Enter your transport expense (in $): "))
   other_expenses = float(input("Enter your other expenses (in $): "))

# Process: Calculate total expenses and savings
                 total_expenses = rent_expense + food_expense + transport_expense +
other_expenses
                 savings = monthly_income - total_expenses
```

```
# Output: Display the results
           print("\n--- Budget Summary ---")
           print(f"Monthly Income: ${monthly_income}")
           print(f"Total Expenses: ${total_expenses}")
           print(f"Remaining Savings: ${savings}")
# Add a suggestion based on savings
          if savings > 0:
            print("Great! You have saved some money this month.")
          else:
         print("You are spending more than your income. Consider reviewing your expenses
```

Explanation

1. **Variables:**
    - monthly_income, rent_expense, etc., store the user's inputs.
    - total_expenses and savings calculate the result.
2. **Data Types:**
    - float is used to handle decimal values (income, expenses).
3. **Identifiers:**
    - Clear and meaningful variable names make the code readable.
4. **Input/Output:**
    - input() gathers user data.
    - print() displays results and suggestions to the user.

Loan EMI Calculator

Calculates the Equated Monthly Installment (EMI) for a loan.

```
# Input: Loan details
print("Welcome to the Loan EMI Calculator!")
loan_amount = float(input("Enter the loan amount (in $): "))
annual_interest_rate = float(input("Enter the annual interest rate (in %): "))
loan_tenure_years = int(input("Enter the loan tenure (in years): "))

# Process: Calculate EMI
monthly_interest_rate = (annual_interest_rate / 100) / 12
loan_tenure_months = loan_tenure_years * 12
emi = loan_amount * monthly_interest_rate * (1 + monthly_interest_rate) **
loan_tenure_months / \
    ((1 + monthly_interest_rate) ** loan_tenure_months - 1)

# Output: Display the EMI
print("\n--- Loan EMI Calculation ---")
print(f"Loan Amount: ${loan_amount}")
print(f"Monthly EMI: ${emi:.2f}")
```

Simple Age Calculator

Calculates the user's age based on the current year and year of birth.

```
# Input: Current year and year of birth
print("Age Calculator")
current_year = int(input("Enter the current year: "))
birth_year = int(input("Enter your year of birth: "))

# Process: Calculate age
age = current_year - birth_year

# Output: Display the user's age
print("\n--- Age Calculation ---")
print(f"You are {age} years old.")
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

Day - 9

Control Flow: If Statements in Python

Control flow refers to the order in which the statements of a program are executed. If statements are one of the fundamental tools in Python used for making decisions and controlling the flow of the program based on certain conditions.

1. What is an If Statement?

An if statement evaluates a condition (an expression that resolves to True or False) and executes a block of code if the condition is True. If the condition is False, the code block is skipped.

2. Syntax of If Statement

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

- Condition: An expression that evaluates to either True or False.
- Indentation: Python uses indentation to define blocks of code. All statements under an if block must be indented.

3. Types of If Statements

a. Simple If Statement

Executes a block of code if the condition is True.

```
Eg: number = 10
    if number > 5:
    print("The number is greater than 5.")
  #Output:
     The number is greater than 5.
```

b. If-Else Statement

Executes one block of code if the condition is True, and another block of code if the condition is False.

```
Eg: number = 3
    if number > 5:
       print("The number is greater than 5.")
    else:
     print("The number is 5 or less.")
```

c. If-Elif-Else Statement

Allows multiple conditions to be checked sequentially. Once a True condition is found, its corresponding block is executed, and the rest are skipped.

```
Eg: number = 8
       if number > 10:
          print("The number is greater than 10.")
       elif number > 5:
          print("The number is greater than 5 but less than or equal to 10.")
       else:
           print("The number is 5 or less.")
     #Output:
            The number is greater than 5 but less than or equal to 10.
```

d. Nested If Statements

An if statement inside another if statement.

```
Eg: number = 12
    if number > 10:
    if number % 2 == 0:
    print("The number is greater than 10 and is even.")
```

4. Key Points About If Statements

1. Conditions must evaluate to a boolean: Conditions use comparison operators (>, <, ==, !=, etc.) or logical operators (and, or, not).
2. Indentation is required: Python relies on indentation to define blocks of code.
3. Only one block executes: In an if-elif-else chain, only the first True block is executed.
4. Empty blocks: Use pass if you need an empty block (placeholder).

```
Eg:     if number > 5:
        pass  # Placeholder for future code
```

5. Logical Operators in If Statements

- and: True if both conditions are true.
- or: True if at least one condition is true.
- not: Reverses the truth value.

Eg:

```
age = 25
if age > 18 and age < 30:
print("You are in your 20s.")
#Output:
  You are in your 20s.
```

6. Examples of If Statements

Example 1: Checking Even or Odd

```
number = int(input("Enter a number: "))
if number % 2 == 0:
   print("The number is even.")
else:
    print("The number is odd.")
```

Example 2: Grading System

```
score = int(input("Enter your score: "))
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
elif score >= 60:
    print("Grade: D")
else:
    print("Grade: F")
```

Example 3: Age Group Classification

```
age = int(input("Enter your age: "))
if age < 18:
print("You are a minor.")
elif age <= 60:
print("You are an adult.")
else:
print("You are a senior citizen.")
```

Example 4: Discount Calculation

```
amount = float(input("Enter the purchase amount: "))
if amount > 1000:
discount = amount * 0.1
print(f"You get a discount of ₹{discount:.2f}.")
else:
print("No discount available.")
```

7. Common Mistakes with If Statements

Missing indentation:

```
if number > 5:
print("This will cause an IndentationError.")  # Error
```

Using assignment (=) instead of comparison (==):

```
if number = 5:  # Error: Use == for comparison
print("Number is 5.")
```

Unreachable code in if-elif-else:

```
if number > 10:
print("Greater than 10.")
elif number > 5:
print("Greater than 5.")  # This will never execute if the first condition is True.
```

Practice Questions

1. Write a program to check if a given year is a leap year.
2. Create a program that asks the user for a temperature in Celsius and outputs whether it's hot, warm, or cold based on the temperature range.
3. Write a program to determine whether a number is positive, negative, or zero.
4. Create a nested if statement to check if a number is even and greater than 50.

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

2. Classify the temperature as "hot", "warm", or "cold" based on the range:

Hot: > 30°C
Warm: 15–30°C
Cold: < 15°C

```
temp_celsius = float(input("Enter temperature in Celsius: "))

if temp_celsius > 30:
    print("It's hot.")
elif 15 <= temp_celsius <= 30:
    print("It's warm.")
else:
    print("It's cold.")
```

```
3. number = float(input("Enter a number: "))
if number > 0:
    print("The number is positive.")
elif number < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

```
4. number = int(input("Enter a number: "))
if number > 50:
    if number % 2 == 0:
        print(f"The number {number} is even and greater than 50.")
    else:
        print(f"The number {number} is greater than 50 but not even.")
else:
    print(f"The number {number} is not greater than 50.")
```

DAY - 10

Loops in Python

Loops in Python are control structures that allow you to execute a block of code repeatedly, either for a fixed number of iterations or until a specific condition is met.

1. Types of Loops in Python

a) for Loop

The for loop in Python is used for iterating over a sequence (like a list, tuple, string, or range).

Syntax:

```
for variable in sequence:
    # Code to execute in each iteration
```

Example:

```
# Iterating through a list              # Output:
numbers = [1, 2, 3, 4]                      # 1
for num in numbers:                         # 2
    print(num)                              # 3
                                            # 4
```

b) while Loop

The while loop continues to execute as long as its condition is True.

Syntax:

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

Example:

```
# Print numbers from 1 to 5
i = 1
while i <= 5:
    print(i)
    i += 1

# Output:
# 1
# 2
# 3
# 4
# 5
```

2. Control Flow in Loops

a) break Statement

The break statement terminates the loop prematurely when a condition is met.

Example:

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

# Output:
# 0
# 1
# 2
# 3
# 4
```

b) continue Statement

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

Example:

```
for i in range(5):
    if i == 2:
        continue
    print(i)

# Output:
# 0
# 1
# 3
# 4
```

c) pass Statement

The pass statement is a placeholder and does nothing. It is used when a statement is syntactically required but no action is needed.

Example:

```
for i in range(3):
    pass  # Placeholder for future code
```

3. Looping Constructs

a) Looping with range()

The range() function generates a sequence of numbers.

Syntax: range(start, stop, step)

Example:

```
# Looping from 1 to 9
for i in range(1, 10):
    print(i)
```

b) Nested Loops

A loop inside another loop is called a nested loop.

Example:

```
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
```

4. Infinite Loops

Infinite loops occur when the terminating condition is never met.

Example:

```
# Dangerous: This will run forever
while True:
    print("This is an infinite loop")
```

Note: To stop an infinite loop during execution, press **Ctrl+C** or manually terminate the program.

5. Loops with else Clause

Both for and while loops can have an else clause, which executes after the loop completes normally (without a break).

Example:

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

```
# Output:
# 0
# 1
#2
# Loop completed
```

6. Practical Use Cases of Loops

a) Calculating Factorials

```
num = 5
factorial = 1
for i in range(1, num + 1):
    factorial *= i
print(f"Factorial of {num} is {factorial}")
```

b) Summing Numbers in a List

```
numbers = [10, 20, 30]
total = 0
for num in numbers:
    total += num
print(f"Sum is {total}")
```

c) Printing Patterns

```
# Printing a right-angled triangle
rows = 4
for i in range(1, rows + 1):
    print('*' * i)
```

```
# Output:
# *
# **
# ***
# ****
```

7. Common Mistakes with Loops

a) Off-by-One Error

Not correctly setting the range can lead to an incorrect number of iterations.

b) Infinite Loop

Missing an update statement in while loops can create infinite loops.

Example (Wrong):

```
i = 0
while i < 5:
    print(i)
# No increment: Infinite loop
```

8. Summary

Loop Type	Description
for loop	Iterates over a sequence or range.
while loop	Repeats as long as a condition is true.
break	Terminates the loop prematurely.
continue	Skips the current iteration and continues.
else	Executes if the loop ends without a break.

Practice Questions

1. Write a program to check if a number is prime using a loop.
2. Print a multiplication table using nested loops.
3. Find the sum of even numbers in a list using a loop.
4. Reverse a string using a loop.
5. Create a pattern like the following:

```
1
12
123
1234
```

1. A number is prime if it is greater than 1 and divisible only by 1 and itself.

```
number = int(input("Enter a number: "))
if number > 1:
    for i in range(2, int(number ** 0.5) + 1):
        if number % i == 0:
            print(f"{number} is not a prime number.")
            break
    else:
        print(f"{number} is a prime number.")
else:
    print(f"{number} is not a prime number.")
```

```
2. rows = int(input("Enter the number of rows for the multiplication table: "))
for i in range(1, rows + 1):
    for j in range(1, rows + 1):
        print(f"{i * j:4}", end=" ")
    print()  # Moves to the next line after each row
```

Sample Output (for rows = 5):

```
1    2    3    4    5
2    4    6    8   10
3    6    9   12   15
4    8   12   16   20
5   10   15   20   25
```

```
3. numbers = [10, 15, 20, 25, 30, 35]
even_sum = 0
for num in numbers:
    if num % 2 == 0:
        even_sum += num
print(f"The sum of even numbers is: {even_sum}")
```

```
4. string = input("Enter a string to reverse: ")
reversed_string = ""
for char in string:
    reversed_string = char + reversed_string  # Add each character to the front
print(f"The reversed string is: {reversed_string}")
```

```
Example:
Input: Python
Output: nohtyP
```

```
5. rows = int(input("Enter the number of rows: "))

for i in range(1, rows + 1):
    for j in range(1, i + 1):
        print(j, end="")
    print()  # Moves to the next line after each row
```

Sample Output (for rows = 4):

```
1
12
123
1234
```

Real Time Application

Bank Account Management System

This program demonstrates a simple bank account system where a user can perform operations like depositing money, withdrawing money, and checking balance. It incorporates rules for identifiers, control flow statements, loops, and input/output in Python.

Day 11

Data Structures

1. **Strings**

Strings are one of the most commonly used data types in Python. A string is a sequence of characters enclosed in either single quotes ('), double quotes ("), or triple quotes (''' or """`).

Key Features of Strings

Immutable: Strings cannot be changed after they are created. Any modification creates a new string.

Sequence Type: Strings are sequences of characters and support operations like indexing, slicing, and iteration.

Example : Creating Strings

```
# Single quotes
str1 = 'Hello'

# Double quotes
str2 = "World"

# Triple quotes (useful for multiline strings)
str3 = '''This is
a multiline
string.'''

print(str1, str2, str3)
```

Accessing Characters

1. Indexing: Access individual characters using their index (starts from 0).

```
Example : string = "Python"
          print(string[0])  # Output: P
          print(string[-1])  # Output: n (negative index for reverse order)
```

2. Slicing: Extract a substring using a range of indices.

```
Example: string = “Python”
         print(string[1:4])  # Output: yth
         print(string[:3])   # Output: Pyt (start defaults to 0)
         print(string[3:])   # Output: hon (end defaults to length of string)
```

String Operations

1. **C**oncatenation: Combine two or more strings using the + operator.

Example :

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

2. Repetition: Repeat a string multiple times using *.

Example :

```
print("Python " * 3)  # Output: Python Python Python
```

3. Membership: Check if a substring exists in a string using in or not in.

Example :

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

4. Length: Find the number of characters in a string using len().

Example :

```
string = "Hello"
print(len(string))  # Output: 5
```

String Formatting

1. Using + Operator:

Example:

```
name = "John"
print("Hello, " + name + "!")  # Output: Hello, John!
```

2. Using format() Method:

Example:

```
age = 25
print("I am {} years old.".format(age))  # Output: I am 25 years old.
```

3. Using f-Strings (Python 3.6+):

Example:

```
name = "Siva"
age = 38
print(f"My name is {name} and I am {age} years old.")  # Output: My
name is Alice and I am 30 years old.
```

Iterating Through Strings

Use a for loop to iterate over each character in a string.

Example :

```
string = "Python"
for char in string:
    print(char)
```

String Comparison

Strings can be compared using comparison operators:

- Equality (==): Checks if two strings are equal.
- Inequality (!=): Checks if two strings are not equal.
- Greater than/less than (>, <): Compares strings lexicographically (based on Unicode values).

Examples

1. Palindrome Check

```
string = input("Enter a string: ")
if string == string[::-1]:
    print("It's a palindrome!")
else:
    print("Not a palindrome.")
```

2. Word Count

```
sentence = input("Enter a sentence: ")
words = sentence.split()
print(f"Number of words: {len(words)}")
```

3. Count Vowels in a String

```
string = input("Enter a string: ").lower()
vowels = "aeiou"
count = 0
for char in string:
    if char in vowels:
        count += 1
print(f"Number of vowels: {count}")
```

4. Reverse a String

```
string = input("Enter a string: ")
reversed_string = string[::-1]
print(f"Reversed string: {reversed_string}")
```

LISTS

Lists are one of the most versatile and commonly used data types in Python. They allow you to store collections of items and perform a variety of operations on them.

What is a List?

A list is a collection of ordered, mutable, and heterogeneous items. Items in a list are enclosed in square brackets [], and they can hold any data type.

Example of a list : Subjects = ["maths", "physics", "chemistry"]

Characteristics of Lists

1. **Ordered:** Items maintain their order of insertion.

```
Example : numbers = [1, 2, 3]
          print(numbers[0])
          # Output: 1
```

2. Mutable: You can modify, add, or remove items from a list after it is created

Example :

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

3. Heterogeneous: Lists can contain different data types.

Example :

```
mixed = [1, "hello", 3.14]
```

4. Dynamic: Lists can grow or shrink as needed.

- Lists grow by using methods like append(), extend(), or insert().
- Lists shrink by using methods like remove(), pop(), or slicing.
- The dynamic nature of lists makes them ideal for handling variable amounts of data efficiently.

Growing a List

- Adding elements to a list dynamically based on user input:

Example :

```
numbers = [] # Ask the user to enter numbers and dynamically add them to the list

print("Enter numbers one by one (type 'done' to finish):")
while True:
    user_input = input("Enter a number: ")
    if user_input.lower() == 'done':
        break
    numbers.append(int(user_input)) # Add the input to the list
print("The dynamically built list is:", numbers)
```

Shrinking a List

Removing elements from a list based on a condition:

Exxample

```
# Start with a list of numbers
numbers = [10, 20, 30, 40, 50]

# Dynamically remove numbers greater than 30
for num in numbers[:]:  # Use a copy of the list to avoid iteration issues
    if num > 30:
        numbers.remove(num)

print("The shrunk list is:", numbers)
```

Combining Growth and Shrinkage

A list can grow and shrink simultaneously based on different conditions:

Example :

```
# Start with an empty list
numbers = []

# Dynamically add numbers
for i in range(5):
    numbers.append(i * 10)

print("List after adding numbers:", numbers)

# Dynamically remove even numbers
for num in numbers[:]:
    if num % 20 == 0:
        numbers.remove(num)

print("List after removing numbers divisible by 20:", numbers)
```

Creating a List

1. Empty List:

```
empty_list = []
```

2. List with Elements:

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

3. Using the list() Constructor:

```
numbers = list((1, 2, 3))
```

4. With a Range:

```
numbers = list(range(5))  # Output: [0, 1, 2, 3, 4]
```

Accessing Items in a List

1. Using Indexing:

- Index starts from 0.
- Negative indexing starts from -1 (last item).

Example :

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Output: apple
print(fruits[-1])  # Output: cherry
```

2. Using Slicing:

- [start:end:step]

Example :

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

Common List Operations

1. **Add Items:**

- append(): Adds a single item at the end.
- extend(): Adds multiple items at the end.
- insert(index, item): Adds an item at a specific position.

Example :

```
fruits = ["apple", "banana"]
fruits.append("cherry")
fruits.extend(["date", "fig"])
fruits.insert(1, "blueberry")
print(fruits)
# Output: ['apple', 'blueberry', 'banana', 'cherry', 'date', 'fig']
```

2.Remove Items:

- remove(item): Removes the first occurrence of the item.
- pop(index): Removes and returns an item at a specific index.
- clear(): Removes all items from the list.

Example:

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
print(fruits)
# Output: ['apple', 'cherry']
fruits.pop(0)
print(fruits)
 # Output: ['cherry']
fruits.clear()
print(fruits)
 # Output: []
```

3. Find Items:

- index(item): Returns the index of the first occurrence of the item.
- count(item): Counts the occurrences of an item.

Example :

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

4. Sort and Reverse:

- sort(): Sorts the list in ascending order (modifies in place).
- sorted(): Returns a sorted list (does not modify the original list).
- reverse(): Reverses the order of items.

Example :

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

List Comprehension

A concise way to create lists using a single line of code.

Examples : 1. **Squaring a Number**

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

2: Filter Even Numbers

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

Common Built-In Functions with Lists

- len(list): Returns the number of items in the list.
- max(list): Returns the largest item in the list.
- min(list): Returns the smallest item in the list.
- sum(list): Returns the sum of all numeric items in the list.
- list(): Converts an iterable to a list.

Iterating Over Lists

1. Using a for Loop:

Example :

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

2. Using enumerate():

Example :

```
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
   print(f"Index {index}: {fruit}")
```

Nested Lists

Lists can contain other lists, creating a multi-dimensional structure.

Example :

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

Copying Lists

1. Shallow Copy:

- Use slicing or the list() constructor.

Example:

```
original = [1, 2, 3]
copy = original[:]
print("Original List:", original) # Output: [1, 2, 3]
print("Copied List:", copy) # Output: [1, 2, 3]
```

2. Deep Copy:

- Use the copy module for nested lists.

Example:

```
import copy
# Nested list
nested_list = [[1, 2, 3], [4, 5], [6, 7, 8]]
# Create a deep copy
deep_copy = copy.deepcopy(nested_list)
# Modify the deep copy
deep_copy[0][0] = 99
# Print both lists
print("Original List:", nested_list)
print("Deep Copy:", deep_copy)
```

Key Points About deepcopy:

- **Independent Copy:**

The copied object (deep_copy) has no shared references with the original object (nested_list), even for nested elements.

- **Use Case:**

Use deepcopy when working with nested objects where changes to one copy shouldn't affect the other.

- **Contrast with Shallow Copy:**

A shallow copy (e.g., nested_list.copy() or copy.copy(nested_list)) only copies the top-level structure, and references to nested objects remain shared.

Tuples in Python

What are Tuples?

A tupleis a collection data type in Python that is:

- Ordered: Elements have a defined order and can be accessed using indices.
- Immutable: Once a tuple is created, its elements cannot be changed, added, or removed.
- Heterogeneous: Can store elements of different data types.

Syntax :

```
# Creating a tuple
my_tuple = (1, 2, 3, "Hello", True)

# Single-element tuple (with a comma)
single_element_tuple = (5,)

# Empty tuple
empty_tuple = ()
```

Key Characteristics

Immutability:

- Tuples cannot be modified after creation.
- However, if a tuple contains a mutable object (like a list), the mutable object can be changed.

Accessing Elements:

- Use indexing to access elements (similar to lists).
- Negative indexing is supported.

Can Contain Duplicate Elements:

- Tuples can store the same value multiple times.

Supports Nesting:

- A tuple can contain other tuples, lists, or dictionaries.

Advantages of Tuples

- Performance: Faster than lists for iteration and access due to immutability.
- Data Integrity: Ensures data remains unchanged (useful for constants).
- Hashable: Can be used as keys in dictionaries if all elements are hashable.

Tuple Operations

1. Accessing Elements

```
# Example tuple
my_tuple = (10, 20, 30, 40)

# Access using indexing
print(my_tuple[1])     # Output: 20
print(my_tuple[-1])    # Output: 40
```

2. Slicing

```
my_tuple = (10, 20, 30, 40, 50)
print(my_tuple[1:4])    # Output: (20, 30, 40)
print(my_tuple[:3])     # Output: (10, 20, 30)
```

3. Tuple Length

```
my_tuple = (1, 2, 3)
print(len(my_tuple))    # Output: 3
```

4. Concatenation and Repetition

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

5. Membership Testing

```
my_tuple = (10, 20, 30)
print(20 in my_tuple)   # Output: True
print(40 not in my_tuple)  # Output: True
```

6. Iterating Through Tuples

```
my_tuple = (10, 20, 30)
for item in my_tuple:
    print(item)
```

7. Tuple Unpacking

```
my_tuple = (1, 2, 3)
a, b, c = my_tuple
print(a, b, c)  # Output: 1 2 3
```

Tuple Methods

Method	Description	Example
.count(x)	Returns the count of x in the tuple	(1, 2, 2).count(2) → 2
.index(x)	Returns the index of the first occurrence of x	(1, 2, 3).index(2) → 1

Immutability Example

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

# Attempt to modify (will raise an error)
# my_tuple[1] = 5  # TypeError: 'tuple' object does not support item
assignment
```

Mutable Objects in Tuples

If a tuple contains a mutable object like a list, the list inside can be changed.

```
my_tuple = (1, [2, 3], 4)
my_tuple[1][0] = 99
print(my_tuple)  # Output: (1, [99, 3], 4)
```

Real-World Applications

1. Storing Constants:

   Example: (PI, E) = (3.14, 2.71)

2. Returning Multiple Values

```
def get_min_max(numbers):
    return min(numbers), max(numbers)
result = get_min_max([3, 5, 1, 7])
print(result)  # Output: (1, 7)
```

3. Using Tuples as Dictionary Keys:

```
coordinates = {(0, 0): "Origin", (1, 2): "Point A"}
print(coordinates[(1, 2)])  # Output: Point A
```

Comparison with Lists

Aspect	**Tuple**	**List**
Mutability	Immutable	Mutable
Performance	Faster	Slower
Use Case	Fixed data, constants	Dynamic data, frequently updated

Practice Questions

1. Create a tuple with mixed data types and access the second element.
2. Write a program to unpack a tuple into separate variables.
3. Count the occurrences of a specific element in a tuple.
4. Create a dictionary with tuples as keys.

Dictionaries in Python

Definition

A dictionary in Python is a collection of key-value pairs, where each key is unique, and values can be any data type. Dictionaries are mutable, meaning their contents can be changed after creation. They are implemented as hash tables, making access to values using keys very fast.

Key Characteristics

1. Key-Value Pairs:

- Each item in a dictionary is stored as a pair: key: value.
- Keys must be immutable (e.g., strings, numbers, or tuples with immutable elements).
- Values can be of any data type and can even be lists, dictionaries, or other objects.

2. Unordered:

- As of Python 3.7, dictionaries maintain insertion order by default. Before Python 3.7, they were unordered.

3. Mutable:
   - You can add, modify, or remove key-value pairs.
4. Dynamic Size:
   - Dictionaries can grow or shrink dynamically as you add or remove elements.
5. Efficient Lookup:
   - Retrieving a value by its key is very fast due to the underlying hash table.

Creating a Dictionary

- Empty Dictionary:

```
my_dict = {}
```

- With Initial Values:

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

Accessing Values

- Using keys:

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

- Using .get() method (avoids KeyError if the key does not exist):

```
print(my_dict.get("city", "Not found"))  # Output: Not found
```

Adding and Modifying Items

- Add a New Key-Value Pair:

```
my_dict["city"] = "New York"
```

- Modify an Existing Key:

```
my_dict["age"] = 30
```

Removing Items

- del Statement:

```
del my_dict["age"]
```

- pop() Method (removes and returns the value):

```
city = my_dict.pop("city")
print(city)  # Output: New York
```

- clear() Method (removes all items):

```
my_dict.clear()
```

Iterating Over Dictionaries

- Keys :

```
for key in my_dict.keys():
    print(key)
```

- Values:

```
for value in my_dict.values():
    print(value)
```

- Key-Value Pairs:

```
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

Dictionary Methods

Method	Description
keys()	Returns a view object of all keys.
values()	Returns a view object of all values.
items()	Returns a view object of key-value pairs.
get(key, default)	Returns the value for the key, or a default value if the key is not found.
pop(key)	Removes the key and returns its value.
popitem()	Removes and returns the last inserted key-value pair.
update()	Updates the dictionary with another dictionary or key-value pairs.
clear()	Removes all items from the dictionary.

Common Use Cases

- Storing Configuration Settings:

```
config = {"theme": "dark", "language": "English", "version": 1.2}
```

- Counting Occurrences:

```
string = "hello"
freq = {}
for char in string:
    freq[char] = freq.get(char, 0) + 1
print(freq)  # Output: {'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

- Representing Tabular Data:

```
employees = {
    "E001": {"name": "Alice", "age": 25, "role": "Engineer"},
    "E002": {"name": "Bob", "age": 30, "role": "Manager"}
}
```

Nested Dictionaries

A dictionary can contain another dictionary as a value:

Example:

```
nested_dict = {
    "person1": {"name": "Alice", "age": 25},
    "person2": {"name": "Bob", "age": 30}
}
print(nested_dict["person1"]["name"])  # Output: Alice
```

Advantages of Dictionaries

- Fast data retrieval using keys.
- Flexible and dynamic for various data structures.
- Useful for representing relationships between data.

Limitations of Dictionaries

- Keys must be immutable and unique.
- Consumes more memory compared to lists and tuples due to hash table storage.

Student Grades Management:

```
students = {
    "Alice": [85, 90, 88],
    "Bob": [78, 81, 74],
    "Charlie": [92, 87, 85]
}

# Adding a new student
students["Diana"] = [88, 89, 90]

# Updating grades for a student
students["Bob"].append(80)

# Displaying each student's average grade
for name, grades in students.items():
    average = sum(grades) / len(grades)
    print(f"{name}'s average grade: {average:.2f}")
```

Sets in Python

A set is an unordered, mutable, and unindexed collection of unique elements. Sets are used to store multiple items in a single variable without duplicates. They are useful for operations like union, intersection, and difference.

Key Features of Sets

- Unordered: The elements in a set have no specific order, and their position can change.
- Unique: A set cannot contain duplicate values. If a duplicate is added, it will be ignored.
- Mutable: You can add or remove items from a set.
- Unindexed: Sets do not support indexing, slicing, or accessing elements by position.

Creating a Set

Sets can be created using curly braces {} or the set() constructor.

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

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

# Creating an empty set
empty_set = set()  # Use set(), NOT {}
print(type(empty_set))  # Output: <class 'set'>
```

Basic Operations on Sets

1. Adding Elements

- Use add() to add a single element.
- Use update() to add multiple elements.

Example :

```
fruits = {"apple", "banana"}
fruits.add("cherry")  # Add a single element
print(fruits)  # Output: {'apple', 'banana', 'cherry'}

fruits.update(["orange", "grape"])  # Add multiple elements
print(fruits)  # Output: {'apple', 'banana', 'cherry', 'orange', 'grape'}
```

2. Removing Elements

- Use remove() or discard() to remove specific elements.
- Use pop() to remove an arbitrary element.
- Use clear() to remove all elements.

Example :

```
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")  # Removes 'banana'; raises an error if not found
print(fruits)  # Output: {'apple', 'cherry'}

fruits.discard("apple")  # Removes 'apple'; does NOT raise an error if not found
print(fruits)  # Output: {'cherry'}

removed_element = fruits.pop()  # Removes an arbitrary element
print(removed_element)  # Output: 'cherry'
print(fruits)  # Output: set() (empty set)

fruits.clear()  # Clears all elements from the set
print(fruits)  # Output: set()
```

3. Checking Membership

- Use the in keyword to check if an element exists in the set.

Example :

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

Set Operations

1. Union

Combines elements from two sets, removing duplicates.

Example :

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
result = set1.union(set2)
print(result)  # Output: {1, 2, 3, 4, 5}
```

2. Intersection

Finds common elements between two sets.

```
result = set1.intersection(set2)
print(result)  # Output: {3}
```

3. Difference

Finds elements in one set but not in the other.

```
result = set1.difference(set2)
print(result)  # Output: {1, 2}
```

4. Symmetric Difference

Finds elements that are in either of the sets but not in both.

```
result = set1.symmetric_difference(set2)
print(result)  # Output: {1, 2, 4, 5}
```

Method	Description
add()	Adds a single element to the set.
update()	Adds multiple elements to the set.
remove()	Removes a specific element; raises an error if not found.
discard()	Removes a specific element; does not raise an error.
pop()	Removes and returns an arbitrary element.
clear()	Removes all elements from the set.
union()	Returns a new set containing all unique elements from both sets.
intersection()	Returns a set of elements common to both sets.
difference()	Returns a set of elements in the first set but not in the second.
symmetric_difference()	Returns a set of elements in either set, but not both.

Iterating Through a Set

```
fruits = {"apple", "banana", "cherry"}
for fruit in fruits:
    print(fruit)
# Output (order may vary):
# apple
# banana
# cherry
```

Applications of Sets

- Removing Duplicates:

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

- Membership Testing: Fast lookups to check if an element exists.
- Mathematical Operations: Perform union, intersection, difference, etc.
- Filtering Data: Use sets to filter out duplicates or unwanted elements in a dataset.

Example: Removing Duplicates from a List

```
names = ["Alice", "Bob", "Alice", "Eve", "Bob"]
unique_names = list(set(names))
print(unique_names)  # Output: ['Alice', 'Bob', 'Eve'] (order may vary)
```

Key Points to Remember

- Sets are unordered; the position of elements can vary.
- Sets only store unique values.
- You cannot have mutable elements (like lists) as set elements, but you can use immutable ones like tuples.

Practice Questions :

- Write a program to find the type of data stored in a variable data.
- What is the difference between a list and a tuple in Python?
- What is the purpose of using end="" in the print() function?
- Write a program to check if a number is positive, negative, or zero.
- Correct the error in this code:

```
age = 18
if age >= 18
   print("You are eligible to vote.")
```

- What is the difference between break and continue in loops?
- Write a program to calculate the sum of all even numbers from 1 to 100.
- Write a program to find the union and intersection of two sets.
- What is the difference between a shallow copy and a deep copy?

Create a program that uses:

- Lists to store items.
- Dictionaries to store item names and prices.
- Loops for iterating over the items.
- Conditional statements to apply discounts based on a condition.
- Input/Output for user interaction.

```
cart = []
items = {"Apple": 30, "Banana": 10, "Cherry": 20}

while True:
    print("Items available:")
    for item, price in items.items():
        print(f"{item}: {price} INR")

    choice = input("Enter an item to add to your cart (or 'done' to finish): ").capitalize()
    if choice == "Done":
        break
    elif choice in items:
        cart.append(choice)
    else:
        print("Item not found!")

total = sum(items[item] for item in cart)
if total > 100:
    total *= 0.9  # Apply a 10% discount

print(f"Items in your cart: {cart}")
print(f"Total amount: {total:.2f} INR")
```