

Python Day-1

Introduction to computer systems, Algorithms and flowcharts, History and application areas of Python, Features of Python, Setting up Python and IDEs, Understanding the Python programming cycle, Simple Programming Problems.

Introduction to Computer Systems

This is a basic outline and notes on the fundamental concepts of computer systems designed for first-year undergraduate students.

1. What is a Computer System?

- A **computer system** is a combination of hardware and software that processes data to perform specific tasks.
 - **Components of a Computer System:**
 1. **Hardware:** The physical components of a computer (CPU, memory, storage, input/output devices).
 2. **Software:** Programs and operating systems that instruct the hardware on what tasks to perform.
 3. **User:** The individual who interacts with the computer system.
-

2. Components of a Computer System

1. Hardware

- **Central Processing Unit (CPU):** Executes instructions and processes data.
 - **Control Unit (CU):** Directs operations within the computer.
 - **Arithmetic Logic Unit (ALU):** Performs arithmetic and logical operations.
- **Memory:**
 - **Primary Memory (RAM, ROM):** Stores data and instructions temporarily.
 - **Secondary Memory (Hard drives, SSDs):** Permanent storage of data.
- **Input Devices:** Keyboard, mouse, microphone, etc.
- **Output Devices:** Monitor, printer, speakers, etc.
- **Storage Devices:** USB drives, optical disks, etc.

2. Software

- **System Software:** Operating systems (Windows, macOS, Linux), utilities.
- **Application Software:** Word processors, web browsers, games.
- **Programming Software:** Tools like compilers, interpreters, and debuggers.

3. Data

- Raw facts and figures processed into meaningful information.
- Represented in binary (0s and 1s).

4. Users

- **End-users** interact with the system to perform tasks.
 - Developers and system administrators maintain and enhance systems.
-

3. Types of Computer Systems

1. **Personal Computers (PCs)**: Used for general-purpose tasks (home, office).
 2. **Servers**: Provide services to other computers or devices in a network.
 3. **Embedded Systems**: Special-purpose computers embedded in devices (e.g., washing machines, ATMs).
 4. **Mainframes and Supercomputers**: Handle large-scale data processing and complex computations.
-

4. Working of a Computer System

- **Input**: Data is entered into the system via input devices.
 - **Processing**: The CPU processes the data based on instructions from the software.
 - **Storage**: Data is stored temporarily in RAM or permanently in secondary storage.
 - **Output**: The processed data is presented to the user via output devices.
-

5. Representation of Data in Computers

- **Binary System**: Computers use the binary number system (0s and 1s).
 - **Bits and Bytes**:
 - 1 bit = smallest unit of data (0 or 1).
 - 1 byte = 8 bits (e.g., 10101100).
 - **Data Units**: Kilobyte (KB), Megabyte (MB), Gigabyte (GB), Terabyte (TB).
-

6. Operating Systems (OS)

- A critical system software that manages hardware, software, and user interaction.
- Functions:
 1. **Process Management**: Scheduling and execution of processes.
 2. **Memory Management**: Allocating and freeing memory.
 3. **File System Management**: Organizing and accessing files.

- 4. **Device Management:** Controlling input/output devices.
 - 5. **Security and Access Control:** Protecting data and resources.
 - Examples: Windows, Linux, macOS, Android.
-

7. Programming and Software Development

- **Programming:** Writing instructions (code) for the computer to execute tasks.
 - **Programming Languages:**
 - **Low-level languages:** Assembly, Machine Language.
 - **High-level languages:** Python, C++, Java.
 - **Compiler vs. Interpreter:**
 - **Compiler:** Translates the entire program into machine code before execution.
 - **Interpreter:** Translates code line-by-line during execution.
-

8. Computer Networks

- **Definition:** A group of interconnected computers that share resources.
 - **Types of Networks:**
 1. **LAN (Local Area Network):** Small geographic area (e.g., office).
 2. **WAN (Wide Area Network):** Large geographic area (e.g., Internet).
 3. **MAN (Metropolitan Area Network):** Covers a city.
 - **Internet:** The largest WAN connecting millions of networks globally.
-

9. Evolution of Computers

- **Generations of Computers:**
 1. **First Generation (1940-1956):** Vacuum tubes, large and slow.
 2. **Second Generation (1956-1963):** Transistors replaced vacuum tubes.
 3. **Third Generation (1964-1971):** Integrated Circuits (ICs).
 4. **Fourth Generation (1971-Present):** Microprocessors and personal computers.
 5. **Fifth Generation (Present and Beyond):** AI and quantum computing.
-

10. Applications of Computer Systems

1. **Education:** E-learning platforms, research tools.
2. **Healthcare:** Diagnostic systems, medical records.
3. **Finance:** Online banking, stock trading.
4. **Entertainment:** Gaming, streaming services.

5. **Business:** Data analytics, ERP systems.
-

ALGORITHM

- Algorithm refers to the logic of the program. It is a step by step description of how to arrive at the solution of the problem.
- An algorithm is a complete, detailed and precise step by step method for solving a problem independently of the hardware and software.

Characteristics of a good algorithm are:

- **Input** – the algorithm receives input.
- **Output** – the algorithm produces output.
- **Finiteness** – the algorithm stops after a finite number of instructions are executed.
- **Precision** – the steps are precisely stated (defined).
- **Uniqueness** – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- **Generality** – the algorithm applies to a set of inputs.






Example: Algorithm to find sum of two numbers:

1. Begin
2. Input the value of A and B
3. $SUM = A + B$
4. Display SUM
5. End.

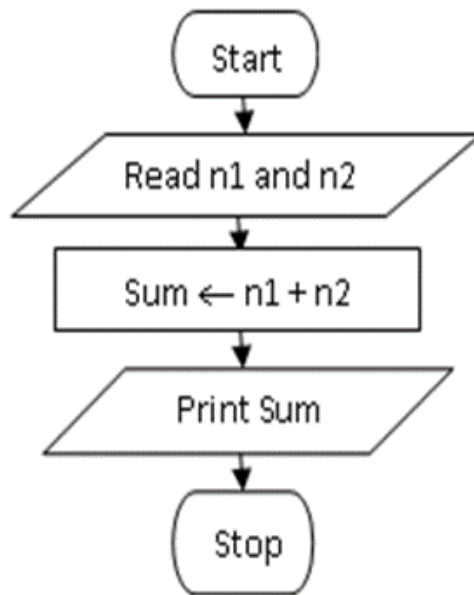
FLOWCHART

- A flowchart is a pictorial representation of an algorithm or process.

Symbols used:

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectagle represents a process
	Decision	A diamond indicates a decision

Example: Flow-Chart to find sum of two numbers



Questions: Write an algorithms and design flow charts for the following:

1. Find average of three numbers.
 2. Swap two numbers using: i. Using third variable ii. Without using third variable
 3. Find area and perimeter of a circle.
 4. Check if a number is divisible by 7 or not.
 5. Among three numbers: a,b and c, find the greatest number.
 6. Check if a number is an Armstrong number or not.
 7. Find Factorial of a number.
-

History of Python

- **Developed By:** Guido van Rossum.
 - **First Released:** In 1991.
 - **Key Philosophy:** Python was designed with readability and simplicity in mind.
 - **Name Origin:** Python is named after the British comedy group "Monty Python," not the snake.
 - **Key Milestones:**
 - **Python 2.0 (2000):** Introduced list comprehensions, garbage collection via reference counting.
 - **Python 3.0 (2008):** Major revision that addressed design flaws and enhanced features (not backward-compatible with Python 2).
-

Application Areas of Python

1. **Web Development:**
 - Frameworks: Django, Flask, FastAPI.
 - Building dynamic websites and APIs.
2. **Data Science and Machine Learning:**
 - Libraries: NumPy, pandas, scikit-learn, TensorFlow, PyTorch.
 - Used for data analysis, visualization, and predictive modeling.
3. **Automation/Scripting:**
 - Automating repetitive tasks and workflows.
4. **Game Development:**
 - Libraries: Pygame.
 - Building simple games.
5. **Scientific Computing:**
 - Libraries: SciPy, SymPy, Matplotlib.
 - Used for mathematical and scientific computations.
6. **Embedded Systems and IoT:**
 - Lightweight frameworks for controlling hardware.
7. **Desktop GUI Applications:**
 - Libraries: Tkinter, PyQt, Kivy.
8. **Cybersecurity and Ethical Hacking:**
 - Libraries: Scapy, Nmap.
 - Used for penetration testing and network analysis.
9. **Blockchain Development:**
 - Libraries: web3.py for working with Ethereum.
10. **Cloud Computing:**
 - Automating cloud environments (e.g., AWS with Boto3).

2. Features of Python

1. **Easy to Learn and Use:**
 - Python has a simple syntax, making it beginner-friendly.
 2. **Open Source:**
 - Freely available and maintained by a large community.
 3. **Interpreted Language:**
 - Executes code line-by-line, making debugging easier.
 4. **Dynamic Typing:**
 - Variable types are inferred at runtime.
 5. **Platform-Independent:**
 - Python code can run on multiple operating systems without modification.
 6. **Extensive Libraries:**
 - Libraries for almost every domain (e.g., NumPy, Flask, Matplotlib).
 7. **Object-Oriented:**
 - Supports object-oriented programming concepts (classes, inheritance).
 8. **Scalable:**
 - Suitable for small scripts and large-scale applications.
 9. **Readable Syntax:**
 - Encourages the use of proper indentation and clean code.
 10. **Integrated Development Environments (IDEs):**
 - Python supports many IDEs like PyCharm, VS Code, and Jupyter Notebook.
-

Setting Up Python and IDEs

Installing Python

1. **Download Python:**
 - Go to the official website: <https://www.python.org/>.
 - Download the latest version for your operating system.
2. **Install Python:**
 - Follow the installation wizard and ensure the "Add Python to PATH" option is selected.
3. **Verify Installation:**

Open a terminal or command prompt and type:

```
python --version
```


or

```
python3 --version
```

Popular IDEs for Python

1. **IDLE:**
 - Comes pre-installed with Python.
 - Lightweight, suitable for beginners.
 2. **PyCharm:**
 - Advanced IDE with features like debugging, testing, and version control.
 3. **Visual Studio Code (VS Code):**
 - Lightweight and extensible editor with Python plugins.
 4. **Jupyter Notebook:**
 - Ideal for data science and interactive computing.
 5. **Spyder:**
 - Popular among scientific computing users.
 6. **Thonny:**
 - Beginner-friendly IDE with simple debugging tools.
-

4. Understanding the Python Programming Cycle

Python Development Workflow

1. **Writing the Code:**
 - Create a `.py` file using an editor or IDE.

Example:

```
print("Hello, World!")
```
2. **Saving the File:**
 - Save the file with a `.py` extension (e.g., `hello.py`).
3. **Running the Code:**
 - Use a terminal or IDE to execute the code:

In a terminal:

```
python hello.py
```

 - In an IDE: Use the "Run" button.
4. **Debugging:**
 - Use tools or manual debugging to fix errors and exceptions.

Example of a common error:

```
print("Hello, World!) # Missing closing quote
```

Fix:

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

5. Testing:

- Test your code with various inputs to ensure reliability.

Example:

```
def add(a, b):  
    return a + b  
print(add(2, 3)) # Output: 5
```

6. Iterating:

- Modify and refine the code based on requirements or errors.

Interpreted Nature of Python

- Python does not require compilation.
- Code is executed line-by-line by the Python interpreter.

Example:

```
a = 5  
b = 10  
print(a + b) # Outputs: 15
```

Different ways to use the `print()` statement in Python.

Printing string	<code>print("Hello, World!")</code>
Printing Variables	<code>name = "Alice"</code> <code>age = 25</code> <code>print(name)</code> <code>print(age)</code>
Printing Multiple Values	<code>name = "Alice"</code> <code>age = 25</code> <code>print("Name:", name, "Age:", age)</code>

Using f-strings (Formatted String Literals) (Python 3.6 and above)	name = "Alice" age = 25 print(f"Name: {name}, Age: {age}")
Using <code>.format()</code> Method	name = "Alice" age = 25 print("Name: {}, Age: {}".format(name, age))
Printing Formatted Numbers	number = 1234567.89123 print(f"Formatted Number: {number:.2f}")
Using String Concatenation	name = "Alice" print("Hello, " + name + "!")
Using <code>sep</code> Parameter	print("Apple", "Banana", "Cherry", sep=", ") print("NIET", "Greater", "Noida", sep="-")
Using <code>end</code> Parameter	print("Hello", end=" ") print("World!")
Printing Escape Sequences	print("Line1\nLine2") # Newline
Printing Raw Strings Use an r or R prefix to print raw strings without interpreting escape sequences.	print(r"Path: C:\Users\Alice\Documents")

Basic Programs:

1. Program to Print "Hello, World!"

```
# we write comments here..
print("Hello, World!")
```

2. Program to Add Two Numbers

```
# Taking input from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))
```

```
# Performing operations
addition = num1 + num2

# Displaying results
print("Addition:", addition)
```

3. Program to Swap Two Numbers Using a Temporary Variable

```
# Taking input from the user
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))

# Swapping using a temporary variable
temp = a
a = b
b = temp

# Displaying swapped values
print("After swapping:")
print("First number:", a)
print("Second number:", b)
```

4. Program to swap two numbers without using a third variable

```
# Taking input from the user
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))

# Swapping without using a third variable
a = a + b
b = a - b
a = a - b

# Displaying swapped values
print("After swapping:")
print("First number:", a)
print("Second number:", b)
```

5. Program to calculate the perimeter and area of a triangle

Explanation

Perimeter:

The perimeter is the sum of all three sides of the triangle:

$$\text{Perimeter} = a + b + c$$

Area (using Heron's formula):

- First, calculate the semi-perimeter: $s = \text{Perimeter} / 2$
- Then calculate the area using: $\text{Area} = \sqrt{s \cdot (s - a) \cdot (s - b) \cdot (s - c)}$

```
import math

# Taking input for the three sides of the triangle
a = float(input("Enter the first side of the triangle: "))
b = float(input("Enter the second side of the triangle: "))
c = float(input("Enter the third side of the triangle: "))

# Calculating the perimeter
perimeter = a + b + c
print("Perimeter of the triangle:", perimeter)

# Using Heron's formula to calculate the area
# Semi-perimeter (s)
s = perimeter / 2

# Area calculation
area = math.sqrt(s * (s - a) * (s - b) * (s - c))

print("Area of the triangle:", area)
```

6. Program to convert Fahrenheit to Celsius

The formula to convert Fahrenheit to Celsius is:

$$c = (f - 32) * 5 / 9$$

```
# Taking temperature input in Fahrenheit
f = float(input("Enter temperature in Fahrenheit: "))

# Conversion formula
c = (f - 32) * 5 / 9
```

```
# Displaying the result
print(f"{f}°F is equal to {c:.2f}°C")
```

7. Program to Calculate Simple Interest

```
principal = float(input("Enter the principal amount: "))
rate = float(input("Enter the rate of interest (in %): "))
time = float(input("Enter the time (in years): "))

simple_interest = (principal * rate * time) / 100

print("The simple interest is:", simple_interest)
```

8. Program to Calculate the Surface Area of a Cylinder

```
import math

r= float(input("Enter the radius of the cylinder: "))
h= float(input("Enter the height of the cylinder: "))

surface_area = 2 * math.pi * r * (r + h)

print("The surface area of the cylinder is:", surface_area)
```

9. Program to Calculate the Hypotenuse of a Right-Angled Triangle

- $\text{hypotenuse} = \sqrt{\text{side1}^2 + \text{side2}^2}$ for exponent(power) we use **

```
import math

side1 = float(input("Enter the first side of the triangle: "))
side2 = float(input("Enter the second side of the triangle: "))

h= math.sqrt(side1**2 + side2**2)

print("The hypotenuse of the triangle is:", h)
```

10. Program to Calculate Compound Interest

```
principal = float(input("Enter the principal amount: "))
rate = float(input("Enter the annual rate of interest (in %): "))
time = float(input("Enter the time (in years): "))

compound_interest = principal * (1 + rate / 100)**time - principal

print("The compound interest is:", compound_interest)
```

List of programs to practice (Home work)

11. Write a program to Find the Average of Three Numbers.
12. Write a program to Calculate Sum of 5 Subjects and Find Percentage.
13. Write a program to find gross salary.
14. Write a program to Calculate Area of Circle.
15. Write a program to Calculate Area of Rectangle.
16. Write a program to Calculate Area of Square.
17. Write a program to Calculate Area and Circumference of Circle.
18. Write a program to Calculate Area of Scalene Triangle.
19. Write a program to Calculate Area of Right angle Triangle.
20. Write a program to find the area of trapezium.

Python Day- 2

(RECAP OF PREVIOUS DAY)

Keywords and identifiers, Variables and data types, Type conversion Operators in Python (Arithmetic, Logical, etc.), Operator precedence and associativity. Problems on above concepts.

Keywords and Identifiers

Keywords

- Keywords are reserved words in Python that have predefined meanings.
- They cannot be used as variable names, function names, or identifiers.

Examples of Python Keywords:

and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, while, with, yield

Identifiers

Identifiers are the names used to identify variables, functions, classes, or other objects.

They must follow these rules:

1. An identifier name may consist of alphabets(A-Z or a-z), digits or underscore.
2. It can not start with a digit.
3. Cannot use reserved keywords.
4. Python is case-sensitive (e.g., **Var** and **var** are different).

Valid and Invalid Identifiers:

```
# Valid identifiers
var1 = 10
_variable = 20
Var = 30
```



```
# Invalid identifiers
1var = 40    # Starts with a digit
class = 50   # Uses a keyword
```

Variables and Data Types

Variables

- Variables are containers for storing data values.
- You don't need to declare the type explicitly in Python; it is dynamically inferred.

Variable Declaration:

```
# Variable declaration and assignment
x = 5
name = "Alice"
pi = 3.14

# Printing variables
print(x)
print(name)
print(pi)
```

Data Types

Python provides several built-in data types:

1. **Numeric:** `int`, `float`, `complex`
2. **Text:** `str`
3. **Boolean:** `bool`
4. **Sequence:** `list`, `tuple`, `range`
5. **Mapping:** `dict`
6. **Set Types:** `set`, `frozenset`
7. **None Type:** `NoneType`

Examples:

```
# Numeric data types
x = 10    # int
y = 3.14  # float
```

```
z = 3 + 4j  # complex

# Text data type
name = "Alice" # str

# Boolean data type
flag = True

# Sequence data type
my_list = [1, 2, 3] # list
my_tuple = (1, 2, 3) # tuple
my_range = range(5) # range

# Mapping data type
details = {"name": "Alice", "age": 25} # dict

# Set types
my_set = {1, 2, 3} # set
my_frozenset = frozenset([1, 2, 3]) # frozenset

# None type
nothing = None
```

Type Conversion

Type conversion is the process of converting one data type into another.

Implicit Type Conversion

Python automatically converts smaller data types to larger ones (e.g., `int` to `float`).

Example:

```
x = 5  # int
y = 2.5 # float
z = x + y
print(z) # Result is a float: 7.5
```

Explicit Type Conversion

Use built-in functions like `int()`, `float()`, `str()`, etc., to explicitly convert data types.

Example:

```
x = "10"  
y = int(x) + 5  
print(y) # Output: 15
```

Operators in Python

Types of Operators

1. **Arithmetic Operators**
 2. **Comparison (Relational) Operators**
 3. **Logical Operators**
 4. **Bitwise Operators**
 5. **Assignment Operators**
 6. **Special Operators** (Identity operator and Membership operator)
-

1. Arithmetic Operators

Used for mathematical operations.

Operator	Description	Example
+	Addition	$5 + 3 = 8$
-	Subtraction	$5 - 3 = 2$
*	Multiplication	$5 * 3 = 15$
/	Division	$5 / 2 = 2.5$
%	Modulus (Remainder)	$5 \% 2 = 1$
//	Floor Division	$5 // 2 = 2$
**	Exponentiation	$5 ** 2 = 25$

Example:

```
x = 10  
y = 3
```

```
print(x + y) # Addition
print(x - y) # Subtraction
print(x * y) # Multiplication
print(x / y) # Division
print(x % y) # Modulus
print(x // y) # Floor Division
print(x ** y) # Exponentiation
```

2. Comparison (Relational) Operators

Used to compare two values.

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

Example:

```
x = 5
y = 3
print(x == y) # False
print(x != y) # True
print(x > y)  # True
print(x < y)  # False
print(x >= y) # True
print(x <= y) # False
```

3. Logical Operators

Used to combine conditional statements.

Operator	Description	Example
and	Returns True if both are True	True and False = False
or	Returns True if one is True	True or False = True
not	Reverses the logical state	not True = False

Example:

```
x = True
y = False
print(x and y) # False
print(x or y)  # True
print(not x)   # False
```

4. Bitwise Operators

Used to perform operations on binary numbers.

Operator	Description	Example
&	Bitwise AND	5 & 3 = 1
	Bitwise OR	5 3 = 7
^	Bitwise XOR	5 ^ 3 = 6
~	Bitwise NOT	~5 = -6
<<	Bitwise Left Shift	5 << 1 = 10
>>	Bitwise Right Shift	5 >> 1 = 2

Example:

```
x = 5 # 0101 in binary
y = 3 # 0011 in binary
print(x & y) # 1 (0001 in binary)
print(x | y) # 7 (0111 in binary)
print(x ^ y) # 6 (0110 in binary)
print(~x)    # -6 (inverts all bits)
```

The bitwise AND (&) operator compares each bit of the two numbers and returns 1 if both bits are 1; otherwise, it returns 0.

Bit position	5 (0101)	3 (0011)	Result (&)
1 (leftmost)	0	0	0
2	1	0	0
3	0	1	0
4 (rightmost)	1	1	1

```
print(x << 1) # 10 (shifts bits to the left)
print(x >> 1) # 2 (shifts bits to the right)
```

Another Example:

Operator	Let a = (92) ₁₀ = (0101 1100) ₂ and b = (14) ₁₀ = (0000 1110) ₂	
&	c = (a & b) = 92 & 14	0101 1100 & 0000 1110 ----- 0000 1100 = (12) ₁₀
	c = (a b) = 92 14	0101 1100 0000 1110 ----- 0101 1110 = (94) ₁₀
^	c = (a ^ b) = 92 14	0101 1100 ^ 0000 1110 ----- 0101 0010 = (82) ₁₀
~	c = ~a = ~92	c = ~(0101 1100) = 1010 0011
<<	c = a<< 1 = 92<< 1	C = 0101 1100 << 1 = 1011 1000 = (184) ₁₀

>>	$c = a \gg 2 = 92 \gg 2$	$C = 0101\ 1100 \gg 2 = 0001\ 0111 = (23)_{10}$
----	--------------------------	---

5. Assignment Operators

Used to assign values to variables.

Operator	Description	Example
=	Assign	$x = 5$
+=	Add and assign	$x += 3$ ($x = 8$)
-=	Subtract and assign	$x -= 3$ ($x = 2$)
*=	Multiply and assign	$x *= 3$ ($x = 15$)
/=	Divide and assign	$x /= 3$ ($x = 1.67$)

$x += 3$ is same as $x = x + 3$

6. Special Operators

- **identity operators** and **membership operators**. These operators are used for specific purposes such as checking object identity or membership in sequences.

A. Identity Operators

Identity operators are used to compare the **memory location** of two objects to check whether they refer to the **same object**.

Operators

Operator	Description	Example
is	Returns True if two variables refer to the same object	$x \text{ is } y$

<code>is not</code>	Returns True if two variables do not refer to the same object	<code>x is not y</code>
---------------------	--	-------------------------

Key Notes

- Objects that have the **same value** may not necessarily have the **same memory location**.
- Mutable objects like lists are stored separately in memory even if they have the same values.

Examples

Example 1: Using ``is`` and ``is not``

```
x = [1, 2, 3]
```

```
y = [1, 2, 3]
```

```
z = x
```

```
print(x is z)           # True (z refers to the same object as x)
```

```
print(x is y)          # False (x and y have same value but different objects)
```

```
print(x is not y)       # True (x and y are not the same object)
```

Example 2: Comparing immutable types

```
a = 10
```

```
b = 10
```

```
print(a is b)          # True (Integers with same value share the same memory)
```

B. Membership Operators

Membership operators are used to check whether a value or object is **present in a sequence** (like a string, list, tuple, or dictionary).

Operators

Operator	Description	Example
<code>in</code>	Returns True if a value is found in the sequence	<code>'a' in 'apple'</code>
<code>not in</code>	Returns True if a value is not found in the sequence	<code>'x' not in 'apple'</code>

Examples

Example 1: Using `in` and `not in` with strings

```
text = "Hello, world!"
print("Hello" in text)          # True ('Hello' is in the string)
print("Python" not in text)     # True ('Python' is not in the string)
```

Example 2: Using `in` and `not in` with lists

```
fruits = ["apple", "banana", "cherry"]
print("apple" in fruits)        # True
print("grape" not in fruits)    # True
```

Example 3: Using `in` with dictionaries (checks keys by default)

```
my_dict = {"name": "Alice", "age": 25}
print("name" in my_dict)        # True
print("Alice" in my_dict)       # False (checks keys, not values)
```

Key Differences Between Identity and Membership Operators

Aspect	Identity Operators (is , is not)	Membership Operators (in , not in)
Purpose	Compare objects' memory locations	Check for membership in a sequence
Example	<code>x is y</code> (checks if <code>x</code> and <code>y</code> are the same object)	<code>'a' in 'apple'</code> (checks if <code>'a'</code> is in the string)
Works On	Any objects (numbers, lists, etc.)	Sequences like strings, lists, tuples, dictionaries

Quick Recap

- **Identity Operators** check if two objects are the same in memory.

- **Membership Operators** check if an element exists in a sequence.

Operator Precedence and Associativity in Python

In Python, **operator precedence** determines the order in which operators are evaluated in an expression. **Associativity** defines the order in which operators of the same precedence level are evaluated (either left-to-right or right-to-left).

Operator Precedence

Operators in Python have different precedence levels. Higher precedence operators are evaluated first. The following table lists operators in descending order of precedence (from highest to lowest):

Precedence Level	Operator	Description
1 (Highest)	<code>()</code>	Parentheses (used for grouping)
2	<code>**</code>	Exponentiation
3	<code>+x, -x, ~x</code>	Unary operators: positive, negative, bitwise NOT
4	<code>*, /, //, %</code>	Multiplication, Division, Floor Division, Modulus
5	<code>+, -</code>	Addition, Subtraction
6	<code><<, >></code>	Bitwise Shift Operators (Left, Right)
7	<code>&</code>	Bitwise AND
8	<code>^</code>	Bitwise XOR
9	<code>`</code>	<code>`</code>
10	<code>==, !=, >, <, >=, <=, is, is not, in, not in</code>	Comparison, Identity, Membership Operators
11	<code>not</code>	Logical NOT
12	<code>and</code>	Logical AND

13 (Lowest)	or	Logical OR
-------------	----	------------

Associativity

When multiple operators with the same precedence appear in an expression, **associativity** determines the order in which they are executed.

Associativity Rules

1. **Left-to-right associativity:** Most operators in Python (e.g., `+`, `-`, `*`, `/`, `//`, `%`, `&`, `|`, etc.) are evaluated from left to right.
 2. **Right-to-left associativity:** Some operators, like the exponentiation operator (`**`) and assignment operators (`=`), are evaluated from right to left.
-

Examples

1. Operator Precedence

```
# Example 1: Exponentiation has higher precedence than multiplication
result = 3 + 2 * 2 ** 2
# Equivalent to: 3 + 2 * (2 ** 2) = 3 + 2 * 4 = 3 + 8 = 11
print(result) # Output: 11
```

```
# Example 2: Parentheses override precedence
result = (3 + 2) * 2 ** 2
# Equivalent to: (3 + 2) * (2 ** 2) = 5 * 4 = 20
print(result) # Output: 20
```

2. Associativity

```
# Example 1: Left-to-right associativity
result = 10 - 5 + 2
# Equivalent to: (10 - 5) + 2 = 5 + 2 = 7
print(result) # Output: 7
```

```
# Example 2: Right-to-left associativity for exponentiation
result = 2 ** 3 ** 2
```

```
# Equivalent to: 2 ** (3 ** 2) = 2 ** 9 = 512
print(result) # Output: 512
```

Parentheses for Clarity

Using parentheses can make expressions clearer and easier to understand, even when you know the precedence rules.

Example: Clarity with Parentheses

```
# Without parentheses
result = 3 + 5 * 2 - 8 / 4
# Equivalent to: 3 + (5 * 2) - (8 / 4) = 3 + 10 - 2 = 11
print(result) # Output: 11

# With parentheses for clarity
result = (3 + (5 * 2)) - (8 / 4)
print(result) # Output: 11
```

Common Mistakes to Avoid

1. Forgetting that ****** (exponentiation) is **right-to-left associative**.

```
print(2 ** 3 ** 2) # Output: 512 (not 64)
```

2. Assuming multiplication (*****) has higher precedence than exponentiation (******).

```
print(2 * 3 ** 2) # Output: 18 (not 36)
```

3. Not using parentheses to group terms when precedence is unclear.
-

Practice Problems

1. What will be the output of the following expression?

```
result = 10 + 2 * 3 ** 2
```

```
print(result)
```

Answer: 28 (Evaluates as $10 + 2 * (3 ** 2) \rightarrow 10 + 2 * 9 \rightarrow 10 + 18$).

2. Rewrite the following expression using parentheses to make the order of operations explicit:

```
result = 4 + 8 / 2 ** 2 * 3
```

Answer: $4 + ((8 / (2 ** 2)) * 3)$

3. Write a program to swap two numbers using bitwise XOR.

```
num1 = 5
num2 = 7
print(f"Before swapping: x = {x}, y = {y}")

# Swapping using XOR
x = x ^ y
y = x ^ y
x = x ^ y

print(f"After swapping: x = {x}, y = {y}")
```

List of programs to practice (Home work)

4. Write a program to find the area of rhombus.
5. Write a program to find the area of parallelogram.
6. Write a program to find the volume and surface area of cube.
7. Write a program to find the volume and surface area of cuboids.
8. Write a program to find the volume and surface area of cylinder.
9. Write a program to find the surface area and volume of a cone.
10. Write a program to find the volume and surface area of sphere.
11. Write a program to find the perimeter of a circle, rectangle and triangle
12. Write a program to Compute Simple Interest.
13. Write a program to Convert Fahrenheit temperature in to Celsius.
14. Write a program to swap the values of two variables.
15. Write a program to swap the values of two variables without using third variable.

Python Day- 3

(RECAP OF PREVIOUS DAY)

Expressions in Python, Strings: Basics, indexing, slicing, and formatting,
Basic string operations and methods. Short discussion on List, tuple, set and dictionary,
Problems on above concepts.

Expressions in Python

Definition:

An expression is a combination of values, variables, operators, and function calls that the Python interpreter can evaluate to produce a result.

Types of Expressions:

1. **Constant Expressions:** Contain only literal values (e.g., `5 + 10`).
2. **Arithmetic Expressions:** Use arithmetic operators to compute numerical values (e.g., `a + b`, `x ** y`).
3. **Logical Expressions:** Use logical operators (`and`, `or`, `not`) to evaluate Boolean conditions.
4. **Relational Expressions:** Compare values using comparison operators (e.g., `a > b`).
5. **Bitwise Expressions:** Perform bitwise operations (e.g., `x & y`).
6. **String Expressions:** Combine string literals or variables using operators (e.g., `"Hello" + " World"`).

Examples of Expressions:

```
# Constant Expression
result = 10 + 20
print(result)           # Output: 30
```

```
# Arithmetic Expression
x = 15
y = 5
result = (x + y) * 2
print(result)           # Output: 40
```

```
# Logical Expression
is_valid = True
is_complete = False
print(is_valid and not is_complete) # Output: True
```

```
# Relational Expression
a = 10
b = 20
print(a < b) # Output: True
```

```
# Bitwise Expression
x = 5 # Binary: 0101
y = 3 # Binary: 0011
print(x & y) # Output: 1 (Binary: 0001)
```

```
# String Expression
greeting = "Hello" + " World"
print(greeting) # Output: Hello World
```

Strings in Python

- Strings in Python are sequences of characters enclosed in single ('), double ("), or triple quotes (' ' ' or " " ").
- Strings are **immutable**, meaning they cannot be changed after creation.

Examples of Strings:

```
# Single-quoted string
greeting = 'Hello'
```

```
# Double-quoted string
name = "Alice"
```

```
# Triple-quoted string (useful for multi-line strings)
multiline_text = """This is
a multi-line
string."""
```

```
print(greeting) # Output: Hello
print(name) # Output: Alice
print(multiline_text) # Output: This is\na multi-line\nstring.
```

String Indexing:

- Strings are indexed from **0 to n-1** (forward indexing) and **-1 to -n** (reverse indexing).

Examples of String Indexing:

```
text = "Python"
```

Forward indexing

```
print(text[0])      # Output: P (1st character)
print(text[3])      # Output: h (4th character)
```

Reverse indexing

```
print(text[-1])     # Output: n (last character)
print(text[-3])     # Output: t (3rd last character)
```

String Slicing:

- Extracting a portion of the string using the slicing operator `:`.
- Syntax: `string[start:end:step]`
 - **start**: Starting index (inclusive, default is 0).
 - **end**: Ending index (exclusive, default is length of string).
 - **step**: Step size (default is 1).

Examples of String Slicing:

```
text = "Python Programming"
```

Basic slicing

```
print(text[0:6])     # Output: Python (characters from index 0 to 5)
print(text[:6])      # Output: Python (same as above)
print(text[7:])      # Output: Programming (from index 7 to end)
```

Using step

```
print(text[0 : 12 : 2]) # Output: Pto rg (every 2nd character from index 0 to 11)
```

Reverse slicing

```
print(text[::-1])    # Output: gnimmargorP nohtyP (reverses the string)
```

String Formatting:

- Formatting allows us to insert variables or values into strings dynamically.

Types of String Formatting:

1. Using `format()` method:

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

2. Using f-strings (Python 3.6+):

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

3. Using `%` operator (old method):

```
name = "Eve"
age = 22
print("My name is %s and I am %d years old." % (name, age))
# Output: My name is Eve and I am 22 years old.
```

3. Basic String Operations and Methods

Basic Operations:

Concatenation: Joining two strings using the `+` operator.

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

1. **Repetition:** Repeating a string using the `*` operator.

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

2. **Membership:** Checking for substring presence using `in` and `not in`.

```
text = "Python Programming"
print("Python" in text)      # Output: True
print("Java" not in text)   # Output: True
```

Common String Methods:

1. `lower()`: Converts the string to lowercase.
2. `upper()`: Converts the string to uppercase.
3. `strip()`: Removes leading and trailing spaces.
4. `split()`: Splits the string into a list of words.
5. `join()`: Joins elements of a list into a string.
6. `replace()`: Replaces a substring with another substring.
7. `find()`: Finds the first occurrence of a substring.
8. `startswith()` and `endswith()`: Check if the string starts or ends with a specific substring.

Examples:

```
text = " Hello Python "
```

```
print(text.lower())    # Output: " hello python "
print(text.upper())    # Output: " HELLO PYTHON "
print(text.strip())    # Output: "Hello Python"
```

```
words = text.split()
print(words)           # Output: ['Hello', 'Python']
```

```
joined = "-".join(words)
print(joined)          # Output: Hello-Python
```

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

```
print(text.find("Python"))                # Output: 8 (starting index of 'Python')
print(text.startswith(" Hello"))          # Output: True
print(text.endswith("Python "))           # Output: True
```

[WE WILL BE DISCUSSING **LIST, TUPLE, SET AND DICTIONARY** IN SHORT HERE, DETAIL DISCUSSION ON THESE TOPICS WILL BE DONE LATER ON DAY 9 AND DAY 10.]

Introduction to List

Definition

- A **list** is an ordered, mutable (modifiable) collection of elements.
- Lists can contain elements of different data types (integers, strings, floats, other lists, etc.).
- Lists are defined using square brackets `[]`.

Key Properties

- **Ordered**: The elements have a specific order.
- **Mutable**: Elements can be added, removed, or modified.
- **Allows duplicates**: Multiple occurrences of the same value are allowed.

Syntax

```
# Creating a list
my_list = [1, 2, 3, "hello", 4.5]
```

Introduction to Tuple

Definition

- A tuple is an ordered, immutable (non-modifiable) collection of elements.
- Tuples can contain elements of different data types.
- Tuples are defined using parentheses `()`.

Key Properties

- **Ordered**: The elements have a specific order.
- **Immutable**: Once defined, elements cannot be changed.
- **Allows duplicates**: Multiple occurrences of the same value are allowed.

```
# Creating a tuple
my_tuple = (1, 2, 3, "hello", 4.5)
```

Introduction to Set

Definition

- A **set** is an unordered, mutable collection of unique elements.
- Sets do not allow duplicate values.
- Sets are defined using curly braces `{ }` or the `set()` function.

Key Properties

- **Unordered**: No guaranteed order of elements.
- **Mutable**: Elements can be added or removed.
- **Unique**: Duplicate elements are not allowed.

Syntax

```
# Creating a set  
my_set = {1, 2, 3, 4}
```

Introduction to Dictionary

Definition

- A **dictionary** is an unordered, mutable collection of key-value pairs.
- Keys are unique, and values can be of any data type.
- Dictionaries are defined using curly braces `{ }` with key-value pairs separated by colons.

Key Properties

- **Unordered**: No guaranteed order of key-value pairs.
- **Mutable**: Keys and values can be added, removed, or modified.
- **Unique Keys**: Keys must be unique, but values can be duplicated.

Syntax

```
# Creating a dictionary
```

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

Practice Problems

1. Convert a string to uppercase and lowercase

```
text = "Python Programming"
print(text.upper())      # Output: PYTHON PROGRAMMING
print(text.lower())      # Output: python programming
```

2. Find the number of occurrences of a substring

```
text = "banana"
print(text.count("a"))   # Output: 3
```

3. Check if a string starts and ends with a specific substring

```
text = "Hello, Python!"
print(text.startswith("Hello"))  # Output: True
print(text.endswith("Python!"))  # Output: True
```

4. Extracting initials from a full name.

```
def get_initials(full_name):
    names = full_name.split()
    initials = "".join([name[0].upper() for name in names])
    return initials

name = "John Doe"
print(get_initials(name))  # Output: JD
```

5. Count vowels in a string.

```
def count_vowels(text):  
    vowels = "aeiouAEIOU"  
    return sum(1 for char in text if char in vowels)  
  
sentence = "Python is fun"  
print(count_vowels(sentence))    # Output: 3
```

6. Reverse words in a sentence.

```
def reverse_words(sentence):  
    words = sentence.split()  
    reversed_sentence = " ".join(reversed(words))  
    return reversed_sentence  
  
text = "Python is fun"  
print(reverse_words(text))    # Output: "fun is Python"
```

7. Replace spaces with hyphens in a string.

```
text = "Python is amazing"  
print(text.replace(" ", "-"))    # Output: "Python-is-amazing"
```

8. Check if a string is a palindrome.

```
def is_palindrome(text):  
    text = text.replace(" ", "").lower()  
    return text == text[::-1]  
  
word = "madam"  
print(is_palindrome(word))    # Output: True
```

List of programs to practice (Home work)

9. Write a program to convert the string from upper case to lower case.
10. Write a program to convert the string from lower case to upper case.
11. Write a program to delete the all consonants from given string.
12. Write a program to count the different types of characters in given string.
13. Write a program to count number of words in a multi word string.
14. Write a program to sort the characters of a string.
15. Write a program for concatenation two strings.
16. Write a program to find the length of a string.
17. Write a program to find the length of a string without using string function.
18. Write a program which prints initial of any name (print RAM for RAM KUMAR).
19. Write a program to check whether a string is palindrome or not.
20. Write a program to sort given names in Lexicographical sorting (Dictionary order).

Python Day- 4

(RECAP OF PREVIOUS DAY)

if statement, if-else statement, Nested if and elif statements, Practical examples and exercises

1. if Statement

Definition:

The `if` statement is used to execute a block of code only if a specified condition evaluates to `True`.

Syntax:

if condition:

 # Code to execute if the condition is True

Example:

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

2. if-else Statement

Definition:

The `if-else` statement provides an alternate block of code to execute when the condition evaluates to `False`.

Syntax:

if condition:

 # Code to execute if the condition is True

else:


```
# Code to execute if the condition is False
```

Example:

```
number = 5
if number % 2 == 0:
    print("The number is even.")
else:
    print("The number is odd.")
# Output: The number is odd.
```

3. Nested if Statements

Definition:

A nested `if` statement is an `if` statement inside another `if` statement. It is used to check multiple levels of conditions.

Syntax:

```
if condition1:
    if condition2:
        # Code to execute if both condition1 and condition2 are True
```

Example:

```
age = 20
citizenship = "India"
if age >= 18:
    if citizenship == "India":
        print("You are eligible to vote in India.")
    else:
        print("You are not an Indian citizen.")
else:
    print("You are not eligible to vote.")
# Output: You are eligible to vote in India.
```

4. elif Statement

Definition:

The `elif` statement stands for "else if" and is used to check multiple conditions in sequence. Once a condition evaluates to `True`, the corresponding block of code is executed, and the rest of the `elif` statements are ignored.

Syntax:

```
if condition1:
    # Code to execute if condition1 is True
elif condition2:
    # Code to execute if condition2 is True
else:
    # Code to execute if none of the above conditions are True
```

Example:

```
marks = 85
if marks >= 90:
    print("Grade: A+")
elif marks >= 80:
    print("Grade: A")
elif marks >= 70:
    print("Grade: B")
elif marks >= 60:
    print("Grade: C")
else:
    print("Grade: F")
# Output: Grade: A
```

Practical Examples and Exercises

1. Check if a number is positive, negative, or zero

```
number = int(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.")
```

2. Find the largest of three numbers

```
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))
c = int(input("Enter the third number: "))

if a > b and a > c:
    print(f"The largest number is {a}.")
elif b > c:
    print(f"The largest number is {b}.")
else:
    print(f"The largest number is {c}.")
```

3. Check if a year is a leap year

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

4. Simple calculator using if-elif-else

```
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))
operation = input("Enter operation (+, -, *, /): ")

if operation == "+":
```

```

    print(f"Result: {num1 + num2}")
elif operation == "-":
    print(f"Result: {num1 - num2}")
elif operation == "*":
    print(f"Result: {num1 * num2}")
elif operation == "/":
    if num2 != 0:
        print(f"Result: {num1 / num2}")
    else:
        print("Division by zero is not allowed.")
else:
    print("Invalid operation.")

```

5. Write a program to check whether a number is divisible by both 5 and 3.

```

number = int(input("Enter a number: "))
if number % 5 == 0 and number % 3 == 0:
    print(f'{number} is divisible by both 5 and 3.')
else:
    print(f'{number} is not divisible by both 5 and 3.')

```

6. Write a program to check whether a character entered by the user is a vowel or consonant.

```

# Input from user
char = input("Enter a single character: ").lower()

# Check if the input is a valid single alphabet
if len(char) == 1 and char.isalpha():
    if char in 'aeiou': # Check if the character is a vowel
        print(f'{char} is a vowel.')
    else:
        print(f'{char} is a consonant.')
else:
    print("Invalid input. Please enter a single alphabet.")

```

7. Write a program to categorize a person's age into child (0-12), teenager (13-19), adult (20-59), and senior (60+).

```

age = int(input("Enter the person's age: "))

if age < 0:
    print("Invalid age. Age cannot be negative.")
elif 0 <= age <= 12:

```

```
print("The person is a child.")
elif 13 <= age <= 19:
    print("The person is a teenager.")
elif 20 <= age <= 59:
    print("The person is an adult.")
else: # age >= 60
    print("The person is a senior.")
```

8. Write a program to calculate the electricity bill based on the following rules:

- First 100 units: \$0.5/unit
- Next 100 units: \$0.75/unit
- Above 200 units: \$1/unit

```
# Input the number of units consumed
units = float(input("Enter the number of units consumed: "))

if units < 0:
    print("Invalid input. Units cannot be negative.")
else:
    # Initialize the bill amount
    bill = 0

    if units <= 100:
        bill = units * 0.5
    elif units <= 200:
        bill = (100 * 0.5) + ((units - 100) * 0.75)
    else:
        bill = (100 * 0.5) + (100 * 0.75) + ((units - 200) * 1)

    # Print the total bill
    print(f"The total electricity bill is: ${bill:.2f}")
```

9. Write a program to determine if a triangle is valid based on the lengths of its sides.

```
# Input the lengths of the three sides of the triangle
side1 = float(input("Enter the length of the first side: "))
side2 = float(input("Enter the length of the second side: "))
side3 = float(input("Enter the length of the third side: "))

# Check if the sides are positive
if side1 <= 0 or side2 <= 0 or side3 <= 0:
    print("Invalid input. Side lengths must be positive.")
else:
    # Check the triangle inequality theorem
    if (side1 + side2 > side3) and (side1 + side3 > side2) and (side2 + side3 > side1):
        print("The triangle is valid.")
    else:
```

```
print("The triangle is not valid.")
```

List of programs to practice (Home work)

10. Write a program to find the greatest between two numbers.
11. Write a program to Accept two Integers and Check if they are Equal.
12. Write a program to Check if a given Integer is Positive or Negative.
13. Write a program to Check if a given Integer is Odd or Even.
14. Write a program to Check if a given Integer is Divisible by 5 or not.
15. Write a program to Accept two Integers and Check if they are Equal.
16. Write a program to find the greatest of three numbers using if...elif.
17. Write a program to find the greatest of three numbers using Nested if without elif.
18. Write a program to convert an Upper case character into lower case and vice-versa.
19. Write a program to check whether an entered year is a leap year or not.
20. Write a Program to check whether an alphabet entered by the user is a vowel or a constant.
21. Write a program to Read a Coordinate Point and Determine its Quadrant.
22. Write a program to Add two Complex Numbers.
23. Write a Program to find roots of a quadratic expression.
24. Write a program to print the day according to the day number entered by the user.
25. Write a program to print color names, if the user enters the first letter of the color name.
26. Write a program to Simulate Arithmetic Calculator using switch.
27. Write a menu driven program for calculating areas of different geometrical figures such as circle, square, rectangle, and triangle.

Python Day- 5

(RECAP OF PREVIOUS DAY)

Purpose and types of loops, while loop, for loop, else with loops, Nested loops, break, continue, and pass statements. Problems on above concepts.

1. Purpose of Loops

Loops are used to execute a block of code multiple times without rewriting it. They:

- Reduce redundancy.
 - Improve code readability and efficiency.
 - Allow iteration over sequences (e.g., lists, tuples, strings) and perform repetitive tasks.
-

2. Types of Loops in Python

a. While Loop

- Executes a block of code as long as the condition is True.

Syntax:

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

Example:

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

b. For Loop

- Iterates over a sequence (e.g., list, tuple, dictionary, string, or range).

Syntax:

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

Example:

```
for i in range(1, 6):  
    print(i)
```

3. else with Loops

- An else block can be used with loops.
- It executes after the loop finishes normally (i.e., without being terminated by a break).

Syntax:

```
for item in sequence:  
    # code block  
else:  
    # code after loop
```

Example:

```
for i in range(5):  
    print(i)  
else:  
    print("Loop completed!")
```

4. Nested Loops

- Loops inside other loops.
- Useful for working with multidimensional data structures.

Example:

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

5. Special Loop Control Statements

a. Break Statement

- Exits the loop immediately, skipping the remaining iterations.

Example:

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

b. Continue Statement

- Skips the current iteration and moves to the next one.

Example:

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

c. Pass Statement

- A placeholder that does nothing and is used when a statement is syntactically required.

Example:

```
for i in range(5):  
    if i == 3:  
        pass # Placeholder for future code  
    print(i)
```

Problems on the Above Concepts

- 1. Print all numbers from 1 to 10 using a while loop.**

```
n = 1  
while n <= 10:  
    print(n)  
    n += 1
```

- 2. Print all numbers from 1 to 10 using a for loop.**

```
for i in range (1,11):  
    print (i)
```

- 3. Iterate over a list of fruits using a for loop.**

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

- 4. Use else with a for loop.**

```
for i in range(3):  
    print(i)
```

```
else:  
    print("Loop completed successfully!")
```

5. Nested loops to print a pattern.

```
for i in range(1, 7):  
    for j in range(1, i + 1):  
        print("*", end="")  
    print() # Newline after each row
```

Or

#Single loop

```
rows = int(input("Enter a number: "))  
for i in range(1, rows + 1):  
    print("*" * i)
```

```
*  
* *  
* * *  
* * * *  
* * * * *  
* * * * * *
```

6. Nested loops to print a pattern.

```
rows = 5  
for i in range(1, rows + 1):  
    for j in range(1, i + 1):  
        print(j, end="")  
    print()
```

```
1  
12  
123  
1234  
12345
```

7. Nested loops to print a pattern.

<pre>rows = 5 num = 1 for i in range(1, rows + 1): for j in range(1, i + 1): print(num, end=" ") num += 1 print()</pre>	<pre>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</pre>
---	--

8. Nested loops to print a pattern.

<pre>rows = 5 for i in range(1, rows + 1): for j in range(65, 65 + i): print(chr(j), end="") print()</pre>	<pre>A AB ABC ABCD ABCDE</pre>
--	--------------------------------

ASCII values of 'A' start at 65

9. Nested loops to print a pattern.

<pre>rows = int(input("Enter a number: ")) for i in range(rows, 0, -1): print("*" * i)</pre>	<pre>* * * * * * * * * * * * * * * * * * *</pre>
--	--

10. Nested loops to print a pattern.

```
rows = int(input("Enter a number: "))
for i in range(1, rows + 1):
    print(" " * (rows - i) + "*" * (2 * i - 1))
```



11. Nested loops to print a pattern.

```
rows = int(input("Enter a number: "))
for i in range(rows, 0, -1):
    print(" " * (rows - i) + "*" * (2 * i - 1))
```



12. Nested loops to print a pattern.

```
rows = 5
# Upper part of the diamond
for i in range(1, rows + 1):
    print(" " * (rows - i) + "*" * (2 * i - 1))

# Lower part of the diamond
for i in range(rows - 1, 0, -1):
    print(" " * (rows - i) + "*" * (2 * i - 1))
```



13. Use break to terminate a loop when a condition is met.

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

14. Use continue to skip an iteration when a condition is met.

```
for i in range(6):
    if i == 3:
        continue
    print(i)
```

15. Use pass as a placeholder in a loop.

```
for i in range(5):
    if i == 2:
        pass # To be implemented later
    print(i)
```

16. Check if a number is prime using a for loop and else.

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

17. Print Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13

```
n = int(input("Enter the number of terms for the Fibonacci series: "))

if n <= 0:
    print("Please enter a positive integer.")
else:
    # Initialize the first two terms
    a, b = 0, 1
    print("Fibonacci Series:")

    for i in range(n):
        print(a, end=" ")
        # Update terms
        a, b = b, a + b
```

18. Program to compute the factorial of a given number

```
num = int(input("Enter a number to compute its factorial: "))

if num < 0:
    print("Factorial is not defined for negative numbers.")
elif num == 0:
    print("The factorial of 0 is 1.")
else:
    factorial = 1
    for i in range(1, num + 1):
        factorial *= i
    print(f"The factorial of {num} is {factorial}.")
```

List of programs to practice (Home work)

19. Write a program to display numbers 1 to 10.

20. Write a program to display all even numbers from 1 to 20
21. Write a program to display all odd numbers from 1 to 20
22. Write a program to print all the Numbers Divisible by 7 from 1 to 100.
23. Write a program to print table of 2.
24. Write a program to print table of 5.
25. Write a program to print table of any number.
26. Write a program to print table of 5 in following format.

$$5 \times 1 = 5$$

$$5 \times 2 = 10$$

$$5 \times 3 = 15$$

27. Write a program to Find the Sum of first 50 Natural Numbers using for Loop.
28. Write a program to calculate factorial of a given number using for loop.
29. Write a program to calculate factorial of a given number using while loop.
30. Write a program to calculate factorial of a given number using do-while loop.
31. Write a program to count the sum of digits in the entered number.
32. Write a program to find the reverse of a given number.
33. Write a program to Check whether a given Number is Perfect Number.
34. Write a program to Print Armstrong Number from 1 to 1000.
35. Write a program to Compute the Value of X^N
36. Write a program to Calculate the value of nCr
37. Write a program to generate the Fibonacci Series
38. Write a program to Print First 10 Natural Numbers
39. Write a program to check whether a given Number is Palindrome or Not
40. Write a program to Check whether a given Number is an Armstrong Number
41. Write a program to Check Whether given Number is Perfect or Not
42. Write a program to check weather a given number is prime number or not.
43. Write a program to print all prime numbers from 50-500
44. Write a program to find the Sum of all prime numbers from 1-1000
45. Write a program to display the following pattern:


```
* * * * *  
* * * * *  
* * * * *  
* * * * *  
* * * * *
```

46. Write a program to display the following pattern:

```
*  
  
* *  
  
* * *  
  
* * * *  
  
* * * * *
```

47. Write a program to display the following pattern:

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5
```

48. Write a program to display the following pattern:

```
1  
2 2  
3 3 3  
4 4 4 4  
5 5 5 5 5
```

49. Write a program to display the following pattern:

```
A  
B B  
C C C  
D D D D  
E E E E E
```

50. Write a program to display the following pattern:

```
* * * * *  
* * * *  
* * *  
* *  
*
```

51. Write a program to display the following pattern:

```
1 2 3 4 5  
1 2 3 4  
1 2 3  
1 2  
1
```

52. Write a program to display the following pattern:

```
      *  
  
    * * *  
  
  * * * * *  
  
* * * * * * *  
  
* * * * * * * *
```

53. Write a program to display the following pattern (Pascal Triangle):

				1				
			1		1			
		1		2		1		
	1		3		3		1	
	1	4		6		4		1
1		5	10		10		5	1
1	6		15	20		15	6	1

41. Write a program to display the following pattern:

```

1
2 3
4 5 6
7 8 9 10

```

54. Write a program to display the following pattern:

```

A
B A B
A B A B A
B A B A B A B

```

55. Write a program to display the following pattern:

```

1
0 1 0
1 0 1 0 1
0 1 0 1 0 1 0

```

56. Write a program to display the following pattern:

```

A B C D E F G F E D C B A
A B C D E F   F E D C B A
A B C D E     E D C B A
A B C D       D C B A
A B C         C B A
A B           B A
A             A

```

57. Write a program to display the following pattern:

```

*****
****  ****
***   ***
**    **
*     *

```

58. Write a program to display the following pattern:

0
01
010
0101
01010

59. Write a program to display the following pattern:

```
1      1
12     21
123    321
1234   4321
12345 54321
```

60. Write a program to display the following pattern:

```
A
B C
D E F
G H I J
K L M N O
```

61. Write a program to display the following pattern:

A
BAB
CBABC
DCBABCD
EDCBABCDE

62. Write a program to display the following pattern:

1A2B3C4D5E
1A2B3C4D
1A2B3C
1A2B
1A

63. Write a program to display the following pattern:

0
1 0 1
2 1 0 1 2
3 2 1 0 1 2 3
4 3 2 1 0 1 2 3 4

64. Write a program to print the following sequence of integers.

0, 5, 10, 15, 20, 25

65. Write a program to print the following sequence of integers.

0, 2, 4, 6, 8, 10

66. Write a program to Find the Sum of A.P Series.

67. Write a program to Find the Sum of G.P Series.

68. Write a program to Find the Sum of H.P Series.

69. Write a program to print the following sequence of integers.

1, 2, 4, 8, 16, 32

70. Write a program to print the following sequence of integers.

-8, -6, -4, -2, 0, 2, 4, 6, 8

71. Write a program to find the Sum of following Series:

$1 + 2 + 3 + 4 + 5 + \dots + n$

72. Write a program to find the Sum of following Series:

$(1*1) + (2*2) + (3*3) + (4*4) + (5*5) + \dots + (n*n)$

73. Write a program to find the Sum of following Series:

$(1) + (1+2) + (1+2+3) + (1+2+3+4) + \dots + (1+2+3+4+\dots+n)$

74. Write a program to find the Sum of following Series:

$1! + 2! + 3! + 4! + 5! + \dots + n!$

75. Write a program to find the Sum of following Series:

$(1^1) + (2^2) + (3^3) + (4^4) + (5^5) + \dots + (n^n)$

76. Write a program to find the Sum of following Series:

$(1!/1) + (2!/2) + (3!/3) + (4!/4) + (5!/5) + \dots + (n!/n)$

77. Write a program to find the Sum of following Series:

$[(1^1)/1] + [(2^2)/2] + [(3^3)/3] + [(4^4)/4] + [(5^5)/5] + \dots + [(n^n)/n]$

78. Write a program to find the Sum of following Series:

$[(1^1)/1!] + [(2^2)/2!] + [(3^3)/3!] + [(4^4)/4!] + [(5^5)/5!] + \dots + [(n^n)/n!]$

79. Write a program to find the Sum of following Series:

$1/2 - 2/3 + 3/4 - 4/5 + 5/6 - \dots$ upto n terms

80. Write a program to print the following Series:

1, 2, 3, 6, 9, 18, 27, 54, ... upto n terms

81. Write a program to print the following Series:

2, 15, 41, 80, 132, 197, 275, 366, 470, 587

82. Write a program to print the following Series:

1, 3, 8, 15, 27, 50, 92, 169, 311

Python Day- 6

(RECAP OF PREVIOUS DAY)

Introduction to functions and function calls, Function arguments (positional, default, keyword, arbitrary arguments), Mutability and immutability, Built-in functions. Problems on above concepts.

Introduction to Functions and Function Calls

Function

A function is a block of reusable code that performs a specific task. Functions help in modularizing code, making it more readable, reusable, and easier to debug.

- A function is a block of organized, reusable code that is used to perform a single, related action.
- Functions provide better modularity for your application and a high degree of code reusing.
- There are two types of functions in Python: Built-in functions and User defined functions
- Python gives many built in functions like `print()`, `len()` etc. But we can also create our own functions. These functions are called user defined functions.

Why Use Functions?

- **Code Reusability:** Write once, use multiple times.
- **Modularity:** Break down complex problems into smaller, manageable parts.
- **Improved Readability:** Clear structure makes the code easier to understand.
- **Ease of Debugging:** Isolate issues within smaller blocks of code.

Syntax of a Function in Python:

```
# Function definition
def function_name(parameters):
```

```
"""Optional docstring to describe the function."""  
# Function body  
return value # Optional  
  
# Function call  
function_name(arguments)
```

Creating a Function

Function for addition of two numbers:

```
def findSum(a,b):           #function definition  
    result = a + b  
    return result  
  
x = 5  
y = 7  
z = findSum(x,y)           #function call  
print(z)
```

12

Note: a,b are called formal arguments while x, y are called actual arguments.

Return Multiple Values

You can also return multiple values from a function. Use the return statement by separating each expression by a comma.

```
def arithmetic(num1,num2):  
    add = num1+num2  
    sub = num1-num2  
    multiply = num1*num2  
    div = num1/num2  
    return add,sub,multiply,div
```

```
a,b,c,d=arithmetic(54,21)  
print(a,b,c,d)
```

75 33 1134 2.5714285714285716

Arguments(or Parameters) in Function

There are the following types of arguments in Python.

1. positional arguments
2. keyword arguments
3. default arguments
4. Variable-length positional arguments
5. Variable-length keyword arguments

Positional Arguments

- During function call, values passed through arguments should be in the order of parameters in the function definition. This is called positional arguments.
- By default, a function must be called with the correct number of arguments.
- Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
def my_function(fname, lname):  
    print(f"My first name is {fname} and the last name is {lname}")  
my_function("Rohan", "Kumar")
```

My first name is Rohan and the last name is Kumar

keyword arguments

- We can also send arguments with the key = value syntax.
- This way the order of the arguments does not matter.
- Keyword arguments are related to the function calls
- The caller identifies the arguments by the parameter name.
- This allows to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

```
def my_function(fname, lname):  
    print(f"My first name is {fname} and the last name is {lname}")  
my_function(lname="Kumar", fname="Mohit")
```

My first name is Mohit and the last name is Kumar

Default Arguments

- Default arguments are values that are provided while defining functions.
- The assignment operator = is used to assign a default value to the argument.
- Default arguments become optional during the function calls.
- If we provide value to the default arguments during function calls, it overrides the default value.
- Default arguments should follow non-default arguments.
- If we call the function without argument, it uses the default value.

```
def my_function(fname, lname="Kumar"):
    print(f"My first name is {fname} and the last name is {lname}")
my_function("Akash")
```

My first name is Akash and the last name is Kumar

Variable-length positional arguments

- Variable-length arguments are also known as Arbitrary arguments.
- If we don't know the number of arguments needed for the function in advance, we can use arbitrary arguments
- For arbitrary positional argument, an asterisk (*) is placed before a parameter in function definition which can hold non-keyword variable-length arguments.
- These arguments will be wrapped up in a tuple. Before the variable number of arguments, zero or more normal arguments may occur.

Syntax:

```
def functionname(*var_args_tuple):
    function_statements
    return [expression]
```

```
def sum_num(num1, *num):
    result = num1
    for i in num:
        result += i
    return result

r = sum_num(10,20,30)
print(f"The sum of numbers is: {r}")

r = sum_num(10,20,30,40,50)
print(f"The sum of numbers is: {r}")
```

The sum of numbers is: 60
The sum of numbers is: 150

Variable length Keyword Arguments

- If you do not know how many keyword arguments that will be passed into your function, add two asterisk (**) before the parameter name in the function definition.
- This way the function will receive a dictionary of arguments, and can access the items accordingly.

```
def my_func(**name):  
    print(f"The first name is {name['fname']} and last name is {name['lname']}")
```

```
my_func(fname="Mohit", lname="Kumar")
```

The first name is Mohit and last name is Kumar

Mutability and Immutability

What is Mutability?

- **Mutable Objects:** Objects that can be changed after creation.
 - Examples: Lists, Dictionaries, Sets
- **Immutable Objects:** Objects that cannot be changed after creation.
 - Examples: Strings, Tuples, Integers, Floats

Examples:

Mutable Example (List):

```
my_list = [1, 2, 3]  
my_list[0] = 10  
print(my_list)           # Output: [10, 2, 3]
```

Immutable Example (String):

```
my_string = "hello"
```

```
# my_string[0] = "H"      # Error: Strings are immutable  
my_string = "Hello"      # Reassignment creates a new object  
print(my_string)         # Output: Hello
```

Functions and Mutability:

- Mutable objects can be modified within functions.
- Immutable objects cannot be modified but can be reassigned.

Example:

```
def modify_list(lst):  
    lst.append(4)      # Modifies the original list  
  
my_list = [1, 2, 3]  
modify_list(my_list)  
print(my_list)        # Output: [1, 2, 3, 4]
```

Built-in Functions

Python provides a wide range of built-in functions that are ready to use. Some commonly used ones include:

Examples:

6. Mathematical Functions:

- a. `abs(x)`: Absolute value of x.
- b. `max(iterable)`: Largest item.
- c. `min(iterable)`: Smallest item.
- d. `sum(iterable)`: Sum of all items.

```
# Example usage of abs()
number = -10
absolute_value = abs(number)
print(f"The absolute value of {number} is {absolute_value}")

# Example usage of max()
numbers_list = [4, 7, 1, 9, 3]
largest_number = max(numbers_list)
print(f"The largest number in the list {numbers_list} is {largest_number}")

# Example usage of min()
smallest_number = min(numbers_list)
print(f"The smallest number in the list {numbers_list} is {smallest_number}")

# Example usage of sum()
total = sum(numbers_list)
print(f"The sum of all numbers in the list {numbers_list} is {total}")
```

7. Type Conversion Functions:

- a. `int(x)`: Converts x to an integer.
- b. `float(x)`: Converts x to a float.
- c. `str(x)`: Converts x to a string.

```
# Example usage of int()
string_number = "42"
integer_value = int(string_number)
print(f"The string '{string_number}' converted to an integer is {integer_value}")

# Example usage of float()
integer_number = 10
float_value = float(integer_number)
print(f"The integer {integer_number} converted to a float is {float_value}")

# Example usage of str()
number = 3.14
string_value = str(number)
print(f"The float {number} converted to a string is '{string_value}'")

# Combining conversions
mixed_string = "123.45"
converted_float = float(mixed_string)
converted_int = int(float(mixed_string)) # Converts to float first, then to int
print(f"The string '{mixed_string}' converted to a float is {converted_float}, and to an
```



```
integer is {converted_int}")
```

8. Input/Output Functions:

- a. `input(prompt)`: Takes input from the user.
- b. `print(value)`: Prints to the console.

```
# Taking input from the user
name = input("Enter your name: ")
age = input("Enter your age: ")

# Printing the input values
print(f"Hello, {name}! You are {age} years old.")
```

9. Iterables and Collections:

- a. `len(x)`: Returns the number of items.
- b. `sorted(iterable)`: Returns a sorted list.
- c. `range(start, stop, step)`: Generates a sequence of numbers.

```
# Example usage of len()
fruits = ["apple", "banana", "cherry", "date"]
length = len(fruits)
print(f"The list of fruits is {fruits}, and it contains {length} items.")

# Example usage of sorted()
unsorted_numbers = [5, 3, 8, 1, 2]
sorted_numbers = sorted(unsorted_numbers)
print(f"The unsorted list is {unsorted_numbers}. The sorted list is {sorted_numbers}.")

# Example usage of range()
start = 1
stop = 10
step = 2
number_sequence = list(range(start, stop, step))
print(f"The range from {start} to {stop} with a step of {step} is: {number_sequence}")
```

10. Other Useful Functions:

- a. `type(x)`: Returns the type of x.

- b. `id(x)`: Returns the memory address of `x`.
- c. `help(obj)`: Displays documentation for `obj`.

```
# Example usage of type()
number = 42
text = "Hello, World!"
print(f"The type of {number} is {type(number)}")
print(f"The type of '{text}' is {type(text)}")

# Example usage of id()
x = 10
y = 10
print(f"The memory address of x (value {x}) is {id(x)}")
print(f"The memory address of y (value {y}) is {id(y)}")

# Example usage of help()
print("\nDocumentation for the built-in len() function:")
help(len)
```

Problems on Functions

1. Write a function to find the factorial of a number.

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

num = 5
result = factorial(num)
print(f"Factorial of {num} is {result}")
```

2. Write a function to check if a number is prime.

```
def is_prime(num):
    if num <= 1:
        return False
    for i in range(2, int(num**0.5) + 1):
```

```
    if num % i == 0:
        return False
    return True

print(is_prime(7))           # Output: True
```

3. Write a function to calculate the nth Fibonacci number using recursion.

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(6))         # Output: 8
```

4. Find sum of digits.

```
def sum_of_digits(n):
    """Calculate the sum of the digits of a number."""
    return sum(int(digit) for digit in str(n))

# Test the function
print(sum_of_digits(1234))   # Output: 10 (1 + 2 + 3 + 4)
```

Explanation: `sum(int(digit) for digit in str(n))`, here each digit in `str(n)` is type casted in `int` and then being summed up.

5. Check Palindrome.

```
def is_palindrome(s):
    return s == s[::-1]

# Test the function
print(is_palindrome("radar"))   # Output: True
print(is_palindrome("hello"))  # Output: False
```

6. Write a function to Find GCD of Two Numbers.

```
def gcd(a, b):  
    while b:  
        a, b = b, a % b          # a=b and b = a%b  
    return a  
  
# Test the function  
print(gcd(12, 18))              # Output: 6
```

7. Count Vowels in a String.

```
def count_vowels(s):  
    vowels = "aeiouAEIOU"  
    return sum(1 for char in s if char in vowels)  
  
# Test the function  
print(count_vowels("hello world"))      # Output: 3
```

8. Check Armstrong Number.

```
def is_armstrong(n):  
    num_str = str(n)  
    num_digits = len(num_str)  
    return sum(int(digit) ** num_digits for digit in num_str) == n  
  
# Test the function  
print(is_armstrong(153))                # Output: True  
print(is_armstrong(123))                # Output: False
```

List of programs to practice (Home work)

9. **Generate a List of Prime Numbers** Write a function `generate_primes(n)` that returns a list of all prime numbers less than or equal to `n`.

10. **Check Armstrong Number** Write a function `is_armstrong(num)` that checks if a number is an Armstrong number. (An Armstrong number of 3 digits is a number such that the sum of its digits raised to the power of 3 equals the number itself, e.g., 153.)
11. Write a function `convert_temperature(temp, to_scale)` that converts a temperature from Celsius to Fahrenheit or Fahrenheit to Celsius based on the value of `to_scale`. Use the formulas:
12. Write a function `find_divisors(num)` that returns a list of all divisors of a given number `num`.
13. Write a function `is_perfect(num)` to check whether a number is a perfect number. (A perfect number is a positive integer that is equal to the sum of its proper divisors, e.g., $6 = 1 + 2 + 3$.)
14. **Check if Two Strings are Anagrams:** Write a function `are_anagrams(s1, s2)` to check if two strings are anagrams of each other.
(Two strings are anagrams if they contain the same characters in different orders, e.g., "listen" and "silent".)
15. **Find the Longest Word in a Sentence:** Write a function `longest_word(sentence)` that takes a sentence as input and returns the longest word in the sentence.
16. **Check for Strong Number:** Write a function `is_strong(num)` to check if a number is a strong number.
(A strong number is a number such that the sum of the factorials of its digits equals the number itself, e.g., $145 = 1! + 4! + 5!$.)
17. Write a program to find the factorial of given number using function.
18. Write a program using function to Find the Average of Three Numbers.
19. Write a program using function to Calculate Sum of 5 Subjects and Find Percentage.
20. Write a program using function to Calculate Area of Circle.
21. Write a program using function to Calculate Area of Rectangle.
22. Write a program using function to Calculate Area and Circumference of Circle.
23. Write a program using function to Calculate Area of Scalene Triangle.
24. Write a program using function to Calculate Area of Right angle Triangle.
25. Write a program using function to find the area of parallelogram.
26. Write a program using function to find the volume and surface area of cube.
27. Write a program using function to find the volume and surface area of cylinder.

Python Day- 7

(RECAP OF PREVIOUS DAY)

Scope rules and namespaces, Recursion, Lambda functions, Functional programming tools: map, filter, reduce. Problems on above concepts.

Scope Rules and Namespaces

Scope

- **Definition:** Scope refers to the region of the code where a variable is accessible.
- **Types of scope:**
 1. **Local Scope:** Variables declared inside a function and accessible only within that function.
 2. **Global Scope:** Variables declared outside all functions and accessible throughout the program.
 3. **Nonlocal Scope:** Used in nested functions to modify variables in the outer (but not global) scope using the `nonlocal` keyword.

Namespaces

- **Definition:** A namespace is a container for variable names, ensuring that variable names are unique and do not conflict.
- **Types of namespaces:**
 1. **Built-in Namespace:** Contains Python's built-in functions like `print()`, `len()`, etc.
 2. **Global Namespace:** Contains global variables.
 3. **Local Namespace:** Contains local variables within a function.

Example:

```
x = 10  # Global variable

def outer_function():
    y = 20  # Nonlocal variable

    def inner_function():
```

```
        nonlocal y
        y += 10
        print(f"Nonlocal y: {y}")

    inner_function()
    print(f"Global x: {x}")

outer_function()
```

Recursion

- A recursive function is a function that calls itself, again and again.
- It means that the function will continue to call itself and repeat its behaviour until some condition is met to return a result.
- All recursive functions share a common structure made up of two parts: base case and recursive case

Example: Calculate the factorial of a number using recursive function.

```
def fact(num):
    if num==0:
        return 1
    else:
        return num * fact(num-1)

n = int(input("Enter a number: "))
f = fact(n)
print(f"The factorial of {n} is {f}")
```

```
Enter a number: 5
The factorial of 5 is 120
```

Example: Print the fibonacci series using recursive function.

```
def fib(num):
    if num==0:
        return 0
    elif num==1:
        return 1
    else:
        return fib(num-1)+fib(num-2)

r = int(input("Enter the range: "))
for i in range(r):
    print(fib(i), end=' ')
```

```
Enter the range: 8
0 1 1 2 3 5 8 13
```

Anonymous/Lambda Function

- Sometimes we need to declare a function without any name. The nameless property function is called an anonymous function or lambda function.
- These functions are called anonymous because they are not declared in the standard manner by using the def keyword
- lambda keyword is used to declare the anonymous function.
- Lambda forms can take any number of arguments but return just one value in the form of an expression.
- Cannot access variables other than those in their parameter list and those in the global namespace.

Syntax:

lambda arg1 ,arg2,.....argn: expression

Example: Program to find the sum of three numbers.


```
result = lambda num1, num2, num3: num1+num2+num3
print(result(10,20,30))
print(result(1,5,6))
```

60

12

- We use lambda functions when we require a nameless function for a short period of time.
- Lambda functions are used along with built-in functions like filter(), map() etc.

filter() Function

The filter() function returns an iterator where the items are filtered through a function to test if the item is accepted or not.

Syntax:

filter(function, sequence)

function – Function argument is responsible for performing condition checking.

sequence – Sequence argument can be anything like list, tuple, string

Example 1: Find even numbers from a list.

```
# Define a List of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Define a function to check if a number is even
def is_even(n):
    if n % 2 == 0:
        return True

# Use filter to get even numbers
even_numbers = filter(is_even, numbers)

# Convert the filter object to a list
even_numbers_list = list(even_numbers)

print(even_numbers_list)
```

[2, 4, 6, 8, 10]

Example 2: Above problem using lambda function

```
# Define a list of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Use filter with a Lambda function to get even numbers
even_numbers = filter(lambda n: n % 2 == 0, numbers)

# Convert the filter object to a list
even_numbers_list = list(even_numbers)

print(even_numbers_list)

[2, 4, 6, 8, 10]
```

Example 3:

```
age = [5, 18, 23, 15, 25, 65, 74, 85, 12]
def myFun(age):
    if age > 18:
        return True
    else:
        return False
adults = filter(myFun, age)
for x in adults:
    print(x, end=" ")

23 25 65 74 85
```

Example 4: Same as example 3 but using lambda function

```
age = [5, 18, 23, 15, 25, 65, 74, 85, 12]
adults = list(filter(lambda x: x>18, age))
print(adults)

[23, 25, 65, 74, 85]
```

map() Function

- The map() function is used to apply some functionality for every element present in the given sequence and generate a new series with a required modification.

- The map() function in Python takes in a function and a list.
- The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Syntax:

map(function, sequence)

function – function argument responsible for applied on each element of the sequence

sequence – Sequence argument can be anything like list, tuple, string

Example: Program to create a new list of cubes of each item of an existing list of Integers using map().

```
num_list = [10, 5, 12, 78, 6, 1, 7, 9]
cube_list = list(map(lambda x:x**3,num_list))
print(cube_list)
```

```
[1000, 125, 1728, 474552, 216, 1, 343, 729]
```

Example: Finding Area of circles whose radii are in a list.

```
radius_list = [0, 10, 20, 30, 40, 100]

area_list = map(lambda r: 3.14 * r * r, radius_list)

# Convert the map object to a list
area_list = list(area_list)

print(area_list)
```

```
[0.0, 314.0, 1256.0, 2826.0, 5024.0, 31400.0]
```

reduce() Function

- The reduce() function is used to minimize sequence elements into a single value by applying the specified condition.
- The reduce() function is present in the functools module. Hence, we need to import it using the import statement before using it.

Syntax:

reduce(function, sequence)

function – function argument responsible for applied on each element of the sequence

sequence – Sequence argument can be anything like list, tuple, string

Example: Find the largest item from a given list.

```
from functools import reduce
num_list = [20, 12, 52, 22, 72, 19, 7]
large = reduce(lambda x,y:x if x>y else y, num_list)
print(large)
```

72

Example: Summing All Elements in a List

```
from functools import reduce

# Define a List of numbers
numbers = [1, 2, 3, 4, 5]

# Use reduce with a Lambda function to sum the numbers
total_sum = reduce(lambda x, y: x + y, numbers)

print(total_sum)
```

15

List of programs to practice (Home work)

1. Scope Rules and Namespaces

1. Write a program to modify a global variable inside a function using the `global` keyword.
 2. Create a nested function where the inner function modifies a variable from the outer function using `nonlocal`.
-

2. Recursion

3. Write a recursive function to compute the sum of the first `n` natural numbers.
 4. Write a recursive function to calculate the factorial of a number.
 5. Implement a recursive function to compute the `n`th Fibonacci number.
 6. Solve the Tower of Hanoi problem for `n` disks.
 7. Write a recursive function to calculate the greatest common divisor (GCD) of two numbers.
 8. Implement a recursive function to compute the sum of digits of a number.
-

3. Lambda Functions

9. Use a lambda function to check if a number is even or odd.
 10. Write a lambda function to calculate the square of a number.
 11. Use a lambda function to find the larger of two numbers.
 12. Write a program to sort a list of dictionaries by a specific key using a lambda function.
 13. Implement a lambda function to calculate the product of two numbers.
 14. Create a lambda function to extract the second element from a tuple.
 15. Write a program to filter out words of length less than 4 from a list using a lambda function.
 16. Write a program to find all numbers divisible by 3 and 5 in a list using a lambda function.
-

Functional Programming Tools (map, filter, reduce)

Map

17. Use `map()` to convert a list of temperatures from Celsius to Fahrenheit.
18. Use `map()` to capitalize the first letter of each word in a list of strings.
19. Write a program to find the square of all numbers in a list using `map()`.
20. Use `map()` with a lambda function to add corresponding elements of two lists.
21. Write a program to append `"_done"` to each string in a list using `map()`.

Filter

22. Use `filter()` to find all even numbers in a list.
23. Write a program to filter out strings of length less than 5 from a list using `filter()`.
24. Use `filter()` to find all prime numbers in a list.
25. Write a program to filter out negative numbers from a list using `filter()`.
26. Use `filter()` to find words starting with a specific letter in a list.

Reduce

27. Use `reduce()` to find the product of all numbers in a list.
 28. Write a program to concatenate all strings in a list into a single string using `reduce()`.
 29. Use `reduce()` to find the maximum number in a list.
 30. Write a program to calculate the sum of the squares of all numbers in a list using `reduce()`.
 31. Use `reduce()` to implement a custom factorial function.
-

Python Day- 8

(RECAP OF PREVIOUS DAY)

Importing and using modules, Writing your own modules, Exploring standard library modules, Understanding packages and dir() function. Problems on above concepts.

Importing and Using Modules

What is a Module?

- A **module** is a file containing Python code (functions, variables, classes) that can be reused in other Python programs.
- Python has built-in modules, third-party modules, and user-defined modules.

How to Import a Module

1. Basic Import:

```
import math

# Using the module
result = math.sqrt(16)
print(result)      # Output: 4.0
```

2. Import Specific Functions or Variables:

```
from math import sqrt, pi

print(sqrt(25))      # Output: 5.0
print(pi)            # Output: 3.141592653589793
```

3. Import with an Alias:

```
import math as m  
  
print(m.sqrt(36))           # Output: 6.0
```

4. Import All from a Module:

```
from math import *  
  
print(sin(0))              # Output: 0.0
```

Note: Avoid using this for large modules, as it can lead to name conflicts.

Writing Your Own Modules

Steps to Create a Module

1. Create a `.py` file with functions, variables, or classes.
2. Import this file into another Python program.

Example

File: `mymodule.py`

```
def add(a, b):  
    return a + b  
  
def subtract(a, b):  
    return a - b  
  
PI = 3.14159
```


Using the Module

```
# Importing the module
import mymodule

# Using functions and variables from the module
print(mymodule.add(5, 3))      # Output: 8
print(mymodule.subtract(10, 4)) # Output: 6
print(mymodule.PI)            # Output: 3.14159
```

Import Specific Functions

```
from mymodule import add, PI

print(add(7, 2))      # Output: 9
print(PI)             # Output: 3.14159
```

Tips for Writing Modules

- Use meaningful names for your functions and variables.
- Add documentation for each function.
- Use `if __name__ == "__main__":` to include test code that won't run when the module is imported.

Example with Test Code:

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

def divide(a, b):
    if b == 0:
        return "Division by zero error"
    return a / b

if __name__ == "__main__":
    print(multiply(4, 5))      # Output: 20
    print(divide(10, 2))      # Output: 5.0
```

Exploring Standard Library Modules

Python provides a rich set of built-in modules known as the **standard library**. Below are some commonly used modules:

os Module

- Used for interacting with the operating system.

Examples:

```
import os

# Get the current working directory
print(os.getcwd())

# List files and directories
print(os.listdir())

# Create a new directory
os.mkdir("new_folder")
```

sys Module

- Provides access to system-specific parameters and functions.

Examples:

```
import sys

# Get command-line arguments
print(sys.argv)

# Exit the program
sys.exit()
```

datetime Module

- Used for working with dates and times.

Examples:

```
from datetime import datetime

# Get the current date and time
now = datetime.now()
print(now)

# Format the date
print(now.strftime("%Y-%m-%d %H:%M:%S"))
```

random Module

- Used for generating random numbers.

Examples:

```
import random

# Generate a random number between 1 and 10
print(random.randint(1, 10))

# Pick a random choice from a list
print(random.choice(["apple", "banana", "cherry"]))
```

math Module

- Provides mathematical functions.

Examples:

```
import math

# Calculate the square root
```

```
print(math.sqrt(49))

# Calculate the sine of a number
print(math.sin(math.pi / 2))
```

Understanding Packages

What is a Package?

- A **package** is a collection of related modules organized into a directory with a special `__init__.py` file.
- The `__init__.py` file can be empty or used to initialize the package.

Creating a Package

1. Create a directory with a meaningful name (e.g., `mypackage`).
2. Add an empty `__init__.py` file to the directory.
3. Add module files to the package directory.

Example Structure:

```
mypackage/
  __init__.py
  module1.py
  module2.py
```

File: `mypackage/module1.py`

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

File: `mypackage/module2.py`

```
def square(n):
    return n * n
```

Using the Package

```
from mypackage import module1, module2

print(module1.greet("Alice"))    # Output: Hello, Alice!
print(module2.square(5))         # Output: 25
```

Understanding the `dir()` Function

What is `dir()`?

- The `dir()` function returns a list of attributes and methods available in a module, class, or object.

Example:

```
import math

# List all attributes and methods in the math module
print(dir(math))
```

Using `dir()` for Custom Modules

```
import mymodule

# List all attributes and methods in the custom module
print(dir(mymodule))
```

Using `dir()` Without Arguments

- Lists the names of all variables, functions, and objects in the current scope.

Example:

```
a = 10
b = "hello"
print(dir())           # Output: ['a', 'b', ...]
```

List of programs to practice (Home work)

1. Write a custom module `mathutils.py` with functions for:
 - Calculating factorial of a number.
 - Checking if a number is prime.
2. Create a package `stringutils` with modules:
 - `string_operations.py`: Functions to reverse a string and check for palindrome.
 - `case_operations.py`: Functions to convert a string to uppercase and lowercase.
3. Write a program to explore the `os` module:
 - Create a directory.
 - List all files in the directory.
 - Delete the created directory.
4. Use the `random` module to:
 - Generate 10 random numbers between 1 and 100.
 - Simulate rolling a dice.
5. Create a package `mathpackage` with the following modules:
 - `arithmetic.py`: Functions for addition, subtraction, multiplication, and division.
 - `geometry.py`: Functions to calculate the area of a circle and rectangle.
6. Write a program to:
 - Import and explore the `sys` module.
 - Print the Python version and platform details.
7. Write a function in a custom module that:
 - Reads a file and counts the frequency of each word.
8. Use the `dir()` function to list all available functions in:
 - The `random` module.

- A custom module you create.
9. Write a script that uses the `datetime` module to:
- Print the current date and time.
 - Calculate the number of days between two dates.
10. Create a package `utilities` with:
- An `__init__.py` file that imports specific functions from its modules.
 - Modules for math operations, string operations, and date operations.

Python Day- 9

(RECAP OF PREVIOUS DAY)

Sequences: Packing and unpacking, Mutable vs. immutable sequences, Strings: Advanced methods, Lists and list comprehensions, Tuples , Problems on above concepts.

Sequences in Python

Sequences are ordered collections of items. Python provides several built-in sequence types, including:

- Strings
- Lists
- Tuples

All sequences share common operations, such as indexing, slicing, and iterating.

Packing and Unpacking

Packing

Packing means assigning multiple values to a single variable.

Packing example

```
packed = 1, 2, 3, 4
```

```
print(packed) # Output: (1, 2, 3, 4)
```

Unpacking

Unpacking allows you to assign elements of a sequence to multiple variables.

Unpacking example

```
x, y, z = 10, 20, 30
```

```
print(x, y, z) # Output: 10 20 30
```

Using * to collect remaining items

```
*a, b = [1, 2, 3, 4]
```

```
print(a, b) # Output: [1, 2, 3] 4
```

Key Notes:

- Number of variables on the left must match the elements unless `*` is used.
- Useful in returning multiple values from a function.

Example with a function:

```
def return_multiple():  
    return 5, 10, 15  
  
val1, val2, val3 = return_multiple()  
print(val1, val2, val3) # Output: 5 10 15
```

Mutable vs. Immutable Sequences

Mutable Sequences

Mutable sequences can be modified after creation. You can change its value.

- Example: **Lists**

```
my_list = [1, 2, 3]
```

```
my_list[1] = 20 # Modifying element
```

```
print(my_list) # Output: [1, 20, 3]
```

Immutable Sequences

Immutable sequences cannot be changed once created.

- Examples: **Strings** and **Tuples**

Strings

```
text = "hello"
```

```
# text[0] = "H" # This will raise an error
```

```
text = "Hello" # Reassignment is allowed
```

Tuples

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

```
# my_tuple[0] = 10 # This will raise an error
```

Comparison of Mutable and Immutable Sequences:

Property	Mutable (Lists)	Immutable (Strings, Tuples)
Modifiable?	Yes	No
Performance	Slower (more overhead)	Faster (fixed structure)
Use Case	When frequent updates are needed	For read-only or fixed data

Strings: Advanced Methods

Strings in Python are immutable sequences of characters. Here are some advanced methods:

Useful String Methods

Method	Description	Example
<code>capitalize()</code>	Converts the first character to uppercase.	<code>"hello".capitalize()</code> → "Hello"

casefold()	Converts the string to lowercase (more aggressive than lower()).	"HELLO".casefold() → "hello"
lower()	Converts all characters to lowercase.	"HELLO".lower() → "hello"
upper()	Converts all characters to uppercase.	"hello".upper() → "HELLO"
title()	Converts the first character of each word to uppercase.	"hello world".title() → "Hello World"
swapcase()	Swaps case of all characters.	"Hello".swapcase() → "hELLO"
find(substring)	Returns the index of the first occurrence of the substring, or -1 if not found.	"hello".find("e") → 1
rfind(substring)	Returns the index of the last occurrence of the substring, or -1 if not found.	"hello".rfind("l") → 3

index(substring)	Like find(), but raises a ValueError if the substring is not found.	"hello".index("e") → 1
count(substring)	Counts occurrences of the substring.	"hello world".count("l") → 3
replace(old, new)	Replaces all occurrences of old with new.	"hello".replace("l", "x") → "hexxo"
split(delimiter)	Splits the string into a list using delimiter.	"a,b,c".split(",") → ["a", "b", "c"]
rsplit(delimiter)	Splits from the right side.	"a,b,c".rsplit(",", 1) → ["a,b", "c"]
splitlines()	Splits the string at line breaks.	"Hello\nWorld".splitlines() → ["Hello", "World"]
strip()	Removes leading and trailing whitespace (or specified characters).	" hello ".strip() → "hello"
lstrip()	Removes leading whitespace (or specified characters).	" hello".lstrip() → "hello"

rstrip()	Removes trailing whitespace (or specified characters).	"hello ".rstrip() → "hello"
startswith(substring)	Returns True if the string starts with the given substring.	"hello".startswith("he") → True
endswith(substring)	Returns True if the string ends with the given substring.	"hello".endswith("lo") → True
join(iterable)	Joins elements of an iterable with the string as a separator.	",".join(["a", "b", "c"]) → "a,b,c"
isalpha()	Returns True if all characters are alphabetic.	"abc".isalpha() → True , "abc123".isalpha() → False
isdigit()	Returns True if all characters are digits.	"123".isdigit() → True , "123abc".isdigit() → False
isalnum()	Returns True if all characters are alphanumeric (letters or digits).	"abc123".isalnum() → True , "abc123".isalnum() → False

isspace()	Returns True if all characters are whitespace.	" ".isspace() → True , " a ".isspace() → False
len()	Returns the length of the string.	len("hello") → 5
format()	Formats the string using placeholders {}.	"Hello, {}".format("World") → "Hello, World"
zfill(width)	Pads the string with zeroes on the left until the specified width is reached.	"42".zfill(5) → "00042"
center(width)	Centers the string in a field of specified width.	"hello".center(10) → " hello "
ljust(width)	Left-justifies the string in a field of specified width.	"hello".ljust(10) → "hello "
rjust(width)	Right-justifies the string in a field of specified width.	"hello".rjust(10) → " hello"

String Formatting

- **f-strings:**

name = "Alice"

```
age = 25
```

```
print(f"Name: {name}, Age: {age}") # Output: Name: Alice, Age: 25
```

Lists and List Comprehensions

Lists

Lists are mutable sequences that can hold mixed data types.

List Methods:

Method	Description	Example
append(x)	Adds an element x to the end of the list.	<pre>lst = [1, 2]; lst.append(3) → [1, 2, 3]</pre>
extend(iterable)	Extends the list by appending all elements from the given iterable.	<pre>lst = [1, 2]; lst.extend([3, 4]) → [1, 2, 3, 4]</pre>
insert(i, x)	Inserts an element x at position i .	<pre>lst = [1, 2]; lst.insert(1, 99) → [1, 99, 2]</pre>

remove(x)	Removes the first occurrence of element x .	<code>lst = [1, 2, 3];</code> <code>lst.remove(2) → [1, 3]</code>
pop([i])	Removes and returns the element at position i (last if i is not provided).	<code>lst = [1, 2, 3];</code> <code>lst.pop(1) → Returns 2, list becomes [1, 3]</code>
clear()	Removes all elements from the list.	<code>lst = [1, 2, 3];</code> <code>lst.clear() → []</code>
index(x, [start, end])	Returns the index of the first occurrence of x in the list.	<code>lst = [1, 2, 3];</code> <code>lst.index(2) → 1</code>
count(x)	Returns the number of occurrences of x in the list.	<code>lst = [1, 2, 2, 3];</code> <code>lst.count(2) → 2</code>
sort(key=None, reverse=False)	Sorts the list in ascending order (or based on a key function).	<code>lst = [3, 1, 2];</code> <code>lst.sort() → [1, 2, 3]</code>

reverse()	Reverses the elements of the list in place.	<pre>lst = [1, 2, 3]; lst.reverse() → [3, 2, 1]</pre>
copy()	Returns a shallow copy of the list.	<pre>lst = [1, 2]; new_lst = lst.copy() → new_lst = [1, 2]</pre>

List Comprehensions

List comprehensions provide a concise(short) way to create lists.

Syntax:

[expression for item in iterable if condition]

Examples:

```
# Create a list of squares

squares = [x**2 for x in range(1, 6)]

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

```
# Filter even numbers

evens = [x for x in range(10) if x % 2 == 0]

print(evens) # Output: [0, 2, 4, 6, 8]
```

Tuples

Tuples are immutable sequences, often used for fixed collections of items.

Tuple Methods

- **tuple.count(value)**: Counts the number of occurrences of a value.
- **tuple.index(value)**: Finds the first index of a value.

Examples:

```
my_tuple = (10, 20, 30, 10)
```

```
print(my_tuple.count(10)) # Output: 2
```

```
print(my_tuple.index(20)) # Output: 1
```

Tuple Packing and Unpacking

```
# Packing

packed = 1, 2, 3

# Unpacking

x, y, z = packed

print(x, y, z) # Output: 1 2 3
```

Problems and Exercises

Problem 1: Packing and Unpacking

Write a function that returns the sum and product of two numbers. Use unpacking to assign the results.

```
def sum_and_product(a, b):
```

```
    return a + b, a * b
```

```
result_sum, result_product = sum_and_product(4, 5)
```

```
print(result_sum, result_product) # Output: 9 20
```

Problem 2: Filtering with List Comprehensions

Create a list of squares for all odd numbers between 1 and 10.

```
odd_squares = [x**2 for x in range(1, 11) if x % 2 != 0]
```

```
print(odd_squares) # Output: [1, 9, 25, 49, 81]
```

Problem 3: Tuple Manipulation

Given a tuple (10, 20, 30, 40), write a function to:

1. Count occurrences of 20.
2. Find the index of 30.

```
my_tuple = (10, 20, 30, 40)
```

```
print(my_tuple.count(20)) # Output: 1
```

```
print(my_tuple.index(30)) # Output: 2
```

Problem 4: Advanced Strings

Write a program to count the number of vowels in a given string.

```
text = "Hello, Python!"
```

```
vowels = [char for char in text.lower() if char in "aeiou"]
```

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

Problem 5: Mutable vs. Immutable

What happens if you try to modify a tuple? Test this with code.

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

```
# my_tuple[0] = 10 # Uncommenting this will raise a TypeError
```

Programs to practice (HW)

1. Sequences: Packing and Unpacking

1. Write a program to demonstrate tuple packing and unpacking with multiple variables.
2. Write a program to swap two variables using tuple unpacking.
3. Create a function that returns multiple values (e.g., sum and product of two numbers) and unpack them in the caller.
4. Write a program to unpack a nested list or tuple and access its inner elements.
5. Demonstrate how to unpack sequences with `*` (e.g., head, `*middle`, tail).

2. Mutable vs. Immutable Sequences

6. Write a program to show the difference between mutable (list) and immutable (tuple) sequences by modifying elements.
7. Create a program to compare memory addresses of mutable vs immutable sequences when their values are changed.
8. Write a program to simulate a shopping cart using a list (mutable) and a tuple (immutable).
9. Demonstrate how slicing works differently in mutable and immutable sequences.
10. Show how concatenation affects both mutable and immutable sequences.

3. Strings: Advanced Methods

11. Write a program to demonstrate the use of `join()`, `split()`, `strip()`, `find()`, and `replace()` methods.

12. Create a program to count the frequency of each word in a given sentence.
13. Write a program to capitalize the first letter of every word in a string and reverse the case of all other letters.
14. Develop a function to find all palindromic substrings in a given string.
15. Implement a program to extract email addresses or URLs from a given text using string methods.

4. Lists and List Comprehensions

16. Write a program to generate a list of squares for numbers from 1 to 10 using list comprehensions.
17. Create a list comprehension to filter out vowels from a given string.
18. Write a program to flatten a 2D list (list of lists) into a single list using list comprehensions.
19. Generate a list of all prime numbers between 1 and 50 using list comprehensions.
20. Write a program to create a dictionary from two lists using list comprehensions.

5. Tuples

21. Write a program to create a tuple, access its elements, and demonstrate its immutability.
22. Develop a program to sort a list of tuples based on the second element of each tuple.
23. Write a program to find the maximum and minimum elements in a tuple of numbers.
24. Create a program to demonstrate the use of tuples as keys in a dictionary.
25. Write a program to find the index of a given element in a tuple.

6. Problems Combining the Above Concepts

26. Write a program to combine two lists into a list of tuples using `zip()`, and then unpack the tuples.

27. Develop a function that takes a list of integers and returns a tuple with the even numbers in one list and odd numbers in another.
 28. Write a program to demonstrate the difference between shallow and deep copies of a list containing tuples.
 29. Create a function that takes a string of numbers separated by commas, converts it to a list of integers, and returns their sum.
 30. Develop a program to convert a list of strings to uppercase and remove duplicates using list comprehensions.
-

Python Day- 10

(RECAP OF PREVIOUS DAY)

Sets and dictionaries, Built-in methods for sets and dictionaries, Looping through data structures expressions. Problems on above concepts.

Sets

Definition:

A set is an unordered collection of unique and immutable elements. Sets are useful for operations involving membership tests, removing duplicates, and mathematical set operations like union, intersection, etc.

Creating Sets

```
# Empty set
```

```
empty_set = set()
```

```
# Non-empty set
```

```
my_set = {1, 2, 3, 4, 5}
```

```
# From a list
```

```
set_from_list = set([1, 2, 2, 3, 4]) # Output: {1, 2, 3, 4}
```

Set Methods

Method	Description	Example
add(element)	Adds an element to the set.	<code>my_set.add(6)</code>
remove(element)	Removes the specified element (throws an error if not found).	<code>my_set.remove(3)</code>
discard(element)	Removes the element, but doesn't throw an error if not found.	<code>my_set.discard(3)</code>
pop()	Removes and returns an arbitrary element.	<code>removed = my_set.pop()</code>
clear()	Removes all elements from the set.	<code>my_set.clear()</code>
union(other_set)	Returns a new set with elements from both sets.	<code>set1.union(set2)</code>

intersection(other_set)	Returns a new set with common elements.	<code>set1.intersection(set2)</code>
difference(other_set)	Returns a new set with elements not in the other set.	<code>set1.difference(set2)</code>
symmetric_difference()	Returns a new set with elements in either, but not both.	<code>set1.symmetric_difference(set2)</code>
issubset(other_set)	Checks if the set is a subset of another.	<code>set1.issubset(set2)</code>
issuperset(other_set)	Checks if the set is a superset of another.	<code>set1.issuperset(set2)</code>

Examples

Example of set operations

set1 = {1, 2, 3}

set2 = {3, 4, 5}

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

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

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

Dictionaries in Python

Definition:

A dictionary is an unordered collection of key-value pairs. Keys are unique and immutable, while values can be mutable.

Creating Dictionaries

```
# Empty dictionary
```

```
empty_dict = {}
```

```
# Non-empty dictionary
```

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

```
# Using dict() constructor
```

```
constructed_dict = dict(name="Bob", age=30)
```

Dictionary Methods

Method	Description	Example
get(key, default)	Returns the value for the key; returns default if not found.	<code>my_dict.get("age", 0)</code>
keys()	Returns a view object of all keys.	<code>list(my_dict.keys())</code>
values()	Returns a view object of all values.	<code>list(my_dict.values())</code>
items()	Returns a view object of all key-value pairs.	<code>list(my_dict.items())</code>
update(other_dict)	Updates the dictionary with key-value pairs from another dictionary.	<code>my_dict.update({"city": "London"})</code>
pop(key)	Removes the key-value pair and returns the value.	<code>my_dict.pop("age")</code>
popitem()	Removes and returns the last key-value pair.	<code>my_dict.popitem()</code>

clear()	Removes all key-value pairs.	<code>my_dict.clear()</code>
----------------	------------------------------	------------------------------

Examples

```
# 1. Using get(key, default)
student = {"name": "John", "age": 25}
print(student.get("name")) # Output: John
print(student.get("grade", "Not available")) # Output: Not available

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

# 3. Using values()
values = student.values()
print(values) # Output: dict_values(['John', 25])

# 4. Using items()
items = student.items()
print(items) # Output: dict_items([('name', 'John'), ('age', 25)])

# 5. Using update(other_dict)
extra_info = {"grade": "A", "city": "New York"}
student.update(extra_info)
print(student) # Output: {'name': 'John', 'age': 25, 'grade': 'A', 'city': 'New York'}

# 6. Using pop(key)
age = student.pop("age")
print(age) # Output: 25
print(student) # Output: {'name': 'John', 'grade': 'A', 'city': 'New York'}

# 7. Using popitem()
last_item = student.popitem()
print(last_item) # Output: ('city', 'New York')
print(student) # Output: {'name': 'John', 'grade': 'A'}

# 8. Using clear()
student.clear()
print(student) # Output: {}
```

Looping Through Data Structures

Examples of Looping

Loop over Sets

```
my_set = {"apple", "banana", "cherry"}  
  
for item in my_set:  
  
    print(item)
```

Loop over Dictionaries

```
dict1 = {"a": 1, "b": 2, "c": 3}  
  
# Loop through keys  
  
for key in dict1:  
  
    print(key)  
  
  
# Loop through values  
  
for value in dict1.values():  
  
    print(value)
```

```
# Loop through key-value pairs

for key, value in dict1.items():

    print(f"{key}: {value}")
```

Combined Example

```
# Nested data structure example

data = {"fruits": {"apple", "banana"}, "veggies": ["carrot", "broccoli"]}

for category, items in data.items():

    print(f"{category}: {items}")
```

Regular Expressions (Regex)

Introduction:

Regular expressions are patterns used for matching strings. They are implemented using the `re` module in Python.

```
import re
```

Common Regex Methods

Method	Description	Example
<code>re.match(pattern, str)</code>	Matches the pattern at the start of the string.	<code>re.match("a.b", "acb")</code>
<code>re.search(pattern, str)</code>	Searches for the pattern anywhere in the string.	<code>re.search("cat", "A cat is here")</code>
<code>re.findall(pattern, str)</code>	Returns all matches of the pattern.	<code>re.findall("\d+", "A123B456")</code>
<code>re.sub(pattern, repl, str)</code>	Replaces matches with the given replacement.	<code>re.sub("\s", "-", "Hello World")</code>

Special Characters

Character	Description
.	Matches any character except newline.
<code>\d</code>	Matches a digit (0-9).
<code>\w</code>	Matches alphanumeric characters.

<code>\s</code>	Matches whitespace characters.
<code>*</code>	Matches 0 or more repetitions.
<code>+</code>	Matches 1 or more repetitions.
<code>?</code>	Matches 0 or 1 repetition.

Examples

```
import re

# Find all digits
print(re.findall(r"\d+", "My age is 25 and yours is 30")) # Output: ['25', '30']

# Check if a string starts with 'Hello'
print(bool(re.match(r"Hello", "Hello World"))) # Output: True

# Replace spaces with underscores
print(re.sub(r"\s", "_", "Hello World")) # Output: Hello_World
```

5. Problems on Above Concepts

Set Problems

1. Write a program to find the union, intersection, and difference between two sets.
2. Create a program to remove duplicates from a list using a set.
3. Implement a program to check if one set is a subset of another.

Dictionary Problems

4. Write a program to count the frequency of characters in a string using a dictionary.
5. Create a dictionary from two lists (keys and values).
6. Implement a program to merge two dictionaries and sort them by key.

Looping Problems

7. Write a program to iterate through a nested dictionary and print all keys and values.
8. Create a program to count the occurrences of each word in a list using a dictionary.

Regex Problems

9. Write a program to extract all email addresses from a given text.
10. Create a program to validate if a given string is a valid phone number (e.g., 123-456-7890).
11. Implement a program to split a paragraph into sentences using regex.