Introduction to Python

Q1.Introduction to Python and its Features.

Python is an interpreted, high-level, generalpurpose programming language created by Guido van Rossum and released in 1991.

Features of python: Simple

and Easy to Learn

interpreted language object-

oriented high level language

Q2.History and evolution of Python.

Python was created by Guido van Rossum and released in 1991. It evolved from the ABC programming language, designed for simplicity and readability. Python 2.0 was released in 2000, introducing key features like garbage collection. In 2008, Python 3.0 was launched with significant changes, but it was not backward compatible. By 2020, Python 2 reached its end of life, and Python 3 has since become the standard, continuing to evolve with improvements in performance, typing, and new features.

Q3.Advantages of using Python over other programming languages.

Python offers simplicity and readability, making it easy to learn and write code. It has an extensive collection of libraries and frameworks, which accelerates development. Python is cross-platform, meaning code runs seamlessly on various operating systems. Its large community ensures robust support and continuous improvements.

Q4.Installing Python and setting up the development environment.

1)Download Python:

Go to the official Python website.

Download the latest stable version of Python for your operating system (Windows, macOS, or Linux).

2)Install Python:

Windows:

Run the downloaded installer.

Make sure to check the box "Add Python to PATH" before clicking "Install Now".

Q5.Writing and executing your first Python program.

1. Create a Python File

   First.py

LAB:

1. **Write a Python program that prints "Hello, World!".**

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

2. **Set up Python on your local machine and write a program to display your name.**
   my_name = "karan modi" print(my_name)
   print(my_name)

2. Programming Style

Q1. Understanding Python's PEP 8 guidelines

PEP 8, the Style Guide for Python Code, is a set of recommendations designed to improve the readability and consistency of Python code. Adhering to these guidelines makes your code easier to understand,

maintain, and collaborate on with others. Here's a breakdown of the key aspects of PEP 8:

Core Principles:

- Readability Counts: The primary goal of PEP 8 is to make Python code more readable. Code is read much more often than it's written, so optimizing for readability is crucial.
- Consistency: Following a consistent style across your projects, and within the Python community in general, reduces cognitive load and makes it easier to understand code.
- A Foolish Consistency is the Hobgoblin of Little Minds: While consistency is important, PEP 8 acknowledges that sometimes, deviating from the guidelines is acceptable if it improves readability or is necessary for compatibility with existing code.

Key Recommendations:

- Indentation:
  - Use 4 spaces per indentation level.
  - Avoid tabs.
- Line Length:
  - Limit all lines to a maximum of 79

characters. ◦ For docstrings or comments, limit lines to 72 characters.

- Blank Lines: ◦ Separate top-level function and class

  definitions with two blank lines. ◦ Separate method definitions inside a class with one blank line. ◦ Use blank lines sparingly inside functions to indicate logical sections.

- Imports: ◦ Imports should usually be on separate lines. ◦ Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants. ◦ Imports should be grouped in the

  following order:

  1. Standard library imports
  2. Related third-party imports
  3. Local application/library specific imports

  ◦ Put a blank line between each group of imports. ◦ Avoid wildcard imports (from <module> import *)

- Whitespace in Expressions and Statements: ◦ Avoid extraneous whitespace immediately

inside parentheses, brackets, or braces. ◦

Avoid whitespace immediately before a

comma, semicolon, or colon. ◦ Use exactly one space around assignment (=) and comparison (==, <, >, !=, <>, <=, >=, in, not in, is, is not) operators. ◦ Don't use spaces around the = sign when used to indicate a keyword argument or a default parameter value.

- Naming Conventions:
  - Functions: lowercase_with_underscores ◦

    Variables: lowercase_with_underscores ◦

    Constants:

    UPPERCASE_WITH_UNDERSCORE

    S ◦ Classes: CamelCase

  - Modules: lowercase_with_underscores ◦

    Packages: lowercase □ Comments:

  - Comments should be complete sentences. ◦ Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. ◦ Inline comments should be used

sparingly. ◦ Docstrings are very important. Write docstrings for all public modules, functions, classes, and methods.

- Docstrings:
  - Use triple double quotes (""") for docstrings. ◦ For one-line docstrings, close the """ on the same line. ◦ Multi-line docstrings should have a summary line, followed by a blank line, followed by a more detailed description.
- Trailing Commas:

- Trailing commas are usually harmless, and sometimes helpful, especially when a list of values is expected to be extended. - Trailing commas are mandatory when creating a tuple of one element.

- Programming Recommendations: - Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, etc.). - Comparisons to singletons like None should always be done with is or is not, never the equality operators. - Use is not rather than not ... is. - Use .startswith() and .endswith() instead of string slicing to check for prefixes or suffixes. - Object type comparisons should always use isinstance() instead of comparing types directly. - Don't compare boolean values to True or False using ==.

Tools for Checking PEP 8 Compliance:

- pycodestyle (formerly pep8): A tool for checking Python code against some of the style conventions in PEP 8.

- flake8: Combines pycodestyle, pyflakes (for detecting errors), and mccabe (for complexity checks).
- pylint: A more comprehensive static analysis tool that checks for a wider range of issues, including style violations.
- black: An opinionated code formatter that automatically formats your code to adhere to PEP 8 (and some other style conventions).

Q2.Indentation, comments, and naming conventions in Python.

When it comes to Python, indentation, comments, and naming conventions are fundamental for writing clean, readable, and maintainable code. Here's a breakdown of each:

1. Indentation:

- Importance:
  - Unlike many other programming languages that use curly braces to define code blocks, Python uses indentation. This makes it a crucial part of the syntax.

    Correct indentation tells the Python interpreter which lines of code belong together.
  - It significantly enhances code readability.

- Rules:
  - Consistent spacing: The most common practice, and the one recommended by PEP 8, is to use 4 spaces for each level of indentation.
  - Avoid mixing tabs and spaces: Mixing tabs and spaces can lead to unexpected errors. It's best to stick to one or the other. Most editors can be set to replace tabs with spaces.
  - Indentation for code blocks: Indent code within control structures (like if, for, while), functions, and classes.
- Example: Python def my_function(x): if x > 5:

print("x is greater than 5")

return True

else:

print("x is 5 or less")

return False

2. Comments:

- Purpose: Comments are used to explain what the code does, making it easier for others (and your

future self) to understand. ◦ They can also be used to temporarily disable code during debugging.

- Types:
  - Single-line comments: Start with a # symbol.
  - Multi-line comments (docstrings): Enclosed in triple quotes (""" or '''). Docstrings are particularly useful for documenting functions, classes, and modules.
- Best practices:
  - Write clear and concise comments. ◦ Keep comments up-to-date with code changes.

    Use docstrings to explain the purpose, parameters, and return values of functions and classes.
- Example:

Python

```
# This is a single-line comment     def
calculate_area(width, height):
    """
    Calculates the area of a rectangle.
```

o

Args:

width: The width of the rectangle.

height: The height of the rectangle.

Returns:

The area of the rectangle.

"""

return width * height

3. Naming Conventions:

Importance:

-

Consistent naming conventions make code more readable and predictable. - They help to convey the purpose and type of variables, functions, and classes.

- PEP 8 recommendations:
  - Variables and functions: Use lowercase with underscores (e.g., my_variable, calculate_total).
  - Classes: Use CamelCase (e.g., MyClass).
  - Constants: Use uppercase with underscores (e.g., MAX_VALUE).
  - Modules and packages: Use lowercase with underscores.
- Example:

Python

```
MAX_COUNT = 100


def process_data(input_list):
result_list = []        for item in
input_list:             # ... process
item ...
        result_list.append(item)
```

```
    return result_list


class DataProcessor:
def __init__(self, data):
        self.data = data
```

Q3. Writing readable and maintainable code.

Writing readable and maintainable code is a crucial skill for any programmer. It's not just about making the code work; it's about making it easy for yourself and others to understand, modify, and debug in the future. Here's a breakdown of key principles and practices:

1. Code Clarity and Readability:

- Meaningful Naming: ◦ Use descriptive names for variables, functions, and classes. Avoid single-letter variables or cryptic abbreviations. ◦ Example: Instead of x, use customerName or totalAmount.
- Consistent Formatting: ◦ Follow established coding style guidelines (like PEP 8 for Python).

  Use consistent indentation, spacing, and line breaks. ◦ Tools like black (for Python) can automate formatting.

- Keep Functions Short and Focused: ○ Each function should have a single, welldefined purpose. ○ Break down complex tasks into smaller, manageable functions. ○ This improves readability and makes code easier to test.
- Avoid Deep Nesting: ○ Excessive nesting of if statements or loops can make code difficult to follow. ○ Simplify logic or use techniques like "guard clauses" to reduce nesting.

2. Effective Documentation:
   - Comments: ○ Use comments to explain complex logic, clarify the purpose of code sections, or provide context. ○ Focus on explaining "why" rather than "what" the code does.

- Don't over-comment obvious code.

- Docstrings: ◦ For functions, classes, and modules, use docstrings to describe their purpose, parameters, and return values. ◦ This provides valuable documentation for users of your code.

3. Code Structure and Organization:
   - Modular Design: ◦ Break down your code into reusable modules or components. ◦ This promotes code reuse and makes it easier to maintain individual parts of the system.
   - Principle of "Don't Repeat Yourself" (DRY): ◦ Avoid duplicating code. Extract common logic into reusable functions or classes. ◦ This reduces the risk of errors and makes it easier to update code.
4. Testing:
   - Write Unit Tests: Create tests to verify that individual functions or components work as expected. ◦ This helps catch bugs early and ensures that code changes don't introduce regressions.
   - Test-Driven Development (TDD):

- Consider writing tests before writing the actual code. - This can help you design cleaner, more testable code.

5. Refactoring:

- Regularly Refactor: - Periodically review your code and make improvements to its structure, readability, and efficiency. - Refactoring should not change the functionality of the code. - Refactoring helps to remove technical debt.

Key Benefits of Readable and Maintainable Code:

- Reduced Debugging Time: Easier to find and fix bugs.
- Increased Collaboration: Easier for other developers to understand and contribute to the code.
- Improved Code Longevity: Easier to maintain and update over time.
- Reduced Risk of Errors: Clear and wellstructured code is less prone to errors.

LAB:

1. Write a Python program that demonstrates the correct use of indentation, comments, and variables following PEP 8 guidelines.
   ANS. def calculate_area(length, width): """ Calculates the area of a rectangle. Args: length: The length of the rectangle. width: The width of the rectangle. Returns: The area of the rectangle. """ area = length * width # Calculate the area return area def main(): """ Main function to demonstrate the calculate_area function. """ rectangle_length = 10 # Example length rectangle_width = 5 # Example width # Calculate and print the area rectangle_area = calculate_area(rectangle_length, rectangle_width) print(f"The area of the rectangle is: {rectangle_area}") # Another example with different values another_length = 8 another_width = 12 another_area = calculate_area(another_length, another_width) print(f"The area of the second rectangle is: {another_area}") if __name__ == "__main__": main() #Additional PEP 8 examples: #Constant naming convention (all uppercase)

-

MAX_VALUE = 100 #Long lines should be broken using implied line continuation inside parentheses, brackets and braces. very_long_variable_name = ("This is a very long string that needs to be broken" "across multiple lines.") #Or using backslashes, but parentheses are preferred. another_very_long_variable_name = \ "This is another very long string that needs to be broken" \ "across multiple lines."

3. Core Python Concepts

Q1.Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

Python has several built-in data types that are essential for handling different kinds of data. Here's a breakdown of the most common ones:

1. Numeric Types:

- Integers (int):
  - These represent whole numbers, both positive and negative, without any decimal points.
    - Examples: 10, -5, 0.

- Floats (float): ◦ These represent real numbers with decimal points.

  Examples: 3.14, -0.001, 2.5e2 (which is 250.0).

2. Sequence Types:

- Strings (str): ◦ These represent sequences of characters. ◦ They are enclosed in single quotes ('...'), double quotes ("..."), or triple quotes ('''...''' or """..."""). ◦ Examples: "hello", 'world', """This is a multi-line string.""".
- Lists (list):
  - ◦ These are ordered, mutable (changeable) sequences of items. ◦ They are enclosed in square brackets ([...]).
  - ◦ Lists can contain items of different data types. ◦ Examples: [1, 2, 3], ['apple', 'banana', 'cherry'], [1, 'hello', 3.14].
- Tuples (tuple): ◦ These are ordered, immutable (unchangeable) sequences of items. ◦ They are enclosed in parentheses ((...)). ◦ Like lists, tuples can contain items of different data

-

  types. ◦ Examples: (1, 2, 3), ('apple', 'banana', 'cherry').

3. Mapping Type:
   - Dictionaries (dict): ◦ These are unordered collections of keyvalue pairs. ◦ They are enclosed in curly braces ({...}). ◦ Keys must be unique and immutable, while values can be of any data type. ◦ Examples: {'name': 'Alice', 'age': 30, 'city': 'New York'}.
4. Set Types:
   - Sets (set): ◦ These are unordered collections of unique elements. ◦ They are enclosed in curly braces ({...}). ◦ Sets automatically remove duplicate elements. ◦ Examples: {1, 2, 3}, {'apple', 'banana', 'cherry'}.

Key Characteristics:

- Mutability:
    - Mutable data types (like lists and dictionaries) can be modified after they are created. ◦ Immutable data types (like integers, floats, strings, and tuples) cannot be changed after they are created.
- Ordering:
    - Ordered data types (like strings, lists, and tuples) maintain the order of their elements. ◦ Unordered data types (like dictionaries and sets) do not guarantee any specific order.

Q2. Python variables and memory allocation

1. Dynamic Typing:

- Python is a dynamically typed language. This means that you don't need to explicitly declare the data type of a variable before assigning a value to it.
- The Python interpreter automatically infers the data type at runtime.
- This flexibility comes with the trade-off of potentially higher memory consumption compared to statically typed languages.

2. Variables as References:

- In Python, variables are essentially references to objects in memory.
- When you assign a value to a variable, you're creating a reference that points to the memory location where the object is stored.
- This is a crucial distinction: variables don't "hold" values directly; they "refer" to them.

3. Memory Allocation:

- Heap Memory: ◦ Python primarily uses the heap for dynamic memory allocation. Objects (including numbers, strings, lists, dictionaries, etc.) are stored in the heap. ◦ The heap is a region of memory where objects can be created and destroyed during program execution.
- Memory Management: ◦ Python employs automatic memory management through a garbage collector.

  This relieves developers from the burden of manually allocating and deallocating memory.

  ◦ Reference Counting:

    - Python uses reference counting to track the number of references to an object.

When an object's reference count drops to zero, it means that no variables are referring to it, and the garbage collector can reclaim the memory.

- Garbage Collection:
  - In addition to reference counting, python also employs a garbage collector that is able to find and clean up circular references.

- Object Interning:
  - Python optimizes memory usage by "interning" certain immutable objects, such as small integers and strings.
  - This means that if multiple variables have the same value, they might refer to the same object in memory, rather than creating separate copies.

4. Memory and Object Identity:

- id() function:
  - The id() function returns the memory address of an object. You can use it to check if two variables refer to the same object.
- is operator:
  - The is operator checks if two variables refer to the same object in memory (i.e.,

if they have the same id()). ◦ The == operator checks if the values of two objects are equal.

Key Considerations:

- Mutable objects (like lists and dictionaries) can be modified in place, which means that changes made through one reference will be visible through other references to the same object.
- Immutable objects (like strings and tuples) cannot be modified in place. When you appear to modify an immutable object, you're actually creating a new object in memory.

Q3. Python operators: arithmetic, comparison, logical, bitwise.

Python operators are symbols that perform operations on values and variables. Here's a breakdown of the main categories:

1. Arithmetic Operators:

These operators perform mathematical calculations.

- + (Addition): Adds two operands. ◦ Example: 5 + 3 equals 8
- - (Subtraction): Subtracts the second operand from the first. ◦ Example: 10 - 4 equals 6

- * (Multiplication): Multiplies two operands. ○ Example: 6 * 7 equals 42
- / (Division): Divides the first operand by the second (returns a float). ○ Example: 10 / 2 equals 5.0
- // (Floor Division): Divides the first operand by the second (returns the integer part of the quotient). ○ Example: 10 // 3 equals 3
- % (Modulus): Returns the remainder of the division.
  - Example: 10 % 3 equals 1
- ** (Exponentiation): Raises the first operand to the power of the second. ○ Example: 2 ** 3 equals 8

2. Comparison Operators:

These operators compare two operands and return a Boolean value (True or False).

- == (Equal to): Checks if two operands are equal. ○ Example: 5 == 5 is True
- != (Not equal to): Checks if two operands are not equal. ○ Example: 5 != 6 is True
- > (Greater than): Checks if the first operand is greater than the second. ○ Example: 8 > 3 is True

- < (Less than): Checks if the first operand is less than the second. ○ Example: 2 < 7 is True
- >= (Greater than or equal to): Checks if the first operand is greater than or equal to the second.
  - Example: 5 >= 5 is True
- <= (Less than or equal to): Checks if the first operand is less than or equal to the second. ○ Example: 4 <= 6 is True

3. Logical Operators:

These operators combine Boolean expressions.

- and: Returns True if both operands are True. ○ Example: True and True is True
- or: Returns True if at least one operand is True. ○ Example: True or False is True
- not: Returns the opposite Boolean value of the operand. ○ Example: not True is False

4. Bitwise Operators:

These operators perform operations on individual bits of integers.

- & (Bitwise AND): Performs a bitwise AND operation.
- | (Bitwise OR): Performs a bitwise OR operation.

- ^ (Bitwise XOR): Performs a bitwise XOR (exclusive OR) operation.
- ~ (Bitwise NOT): [1] Performs a bitwise [2] NOT (inversion) operation.

1. buildbytes.in

buildbytes.in

2. github.com

github.com

- << (Left shift): Shifts the bits to the left.
- >> (Right shift): Shifts the bits to the right.

Example of Bitwise Operators:

Python a = 10 # Binary: 1010

b = 4 # Binary: 0100

print(a & b) # Bitwise AND: 0 (Binary: 0000)

print(a | b) # Bitwise OR: 14 (Binary: 1110)

print(a ^ b) # Bitwise XOR: 14 (Binary: 1110)

print(~a) # Bitwise NOT: -11 print(a << 2) #

left shift. 40 print(a >> 2) # right shift 2

3. Core Python Concepts

Q1. Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

1. Integers (int):
   - These represent whole numbers, both positive and negative, without any decimal points.
   - Examples: -10, 0, 5, 100
2. Floats (float):
   - These represent real numbers with decimal points.
   - Examples: 3.14, -0.5, 2.0
3. Strings (str):
   - These represent sequences of characters, enclosed in single or double quotes.
   - Examples: "hello", 'world', "123"
4. Lists (list):

- These are ordered collections of items, and they are mutable (meaning you can change them).
- Lists can contain items of different data types.
- Examples: [1, 2, "three", 4.0]

5. Tuples (tuple):
   - These are also ordered collections of items, but they are immutable (meaning you cannot change them after they are created).
   - Examples: (1, 2, "three")
6. Dictionaries (dict):
   - These are collections of key-value pairs.
   - Keys must be unique and immutable (like strings or numbers), while values can be of any data [1] type.

1. medium.com

medium.com

   - Examples: {"name": "Alice", "age": 30}
7. Sets (set):
   - These are unordered collections of unique items.

- Sets automatically remove duplicate values.
- Examples: {1, 2, 3}

Why Data Types Matter:

- Memory Management: Data types tell the computer how much memory to allocate for each piece of data.
- Operations: They determine what operations can be performed on the data. For example, you can add two integers, but you can't directly add an integer and a string.
- Error Prevention: Using the correct data types helps prevent errors in your code. Q2. Python variables and memory allocation

1. Python's Dynamic Typing:

- Unlike languages like C or Java, Python is dynamically typed. This means you don't explicitly declare the data type of a variable.

  The interpreter determines the type at runtime.
- This flexibility comes with Python managing memory allocation behind the scenes.

2. Variables as Names/References:

- In Python, variables are essentially "names" or "references" that point to objects in memory.

- When you assign a value to a variable, you're creating a reference to an object that holds that value.
- If you reassign the variable, you're changing the reference to point to a different object.

3. Memory Allocation:
   - Python uses a private heap to store objects.
   - The Python memory manager handles the allocation and deallocation of memory within this heap.
   - Key aspects of Python's memory allocation include:
     - Dynamic Memory Allocation: Python primarily uses dynamic memory allocation. This means memory is allocated as needed during program execution. This allows for great flexibility.
     - Garbage Collection: Python has an automatic garbage collector that reclaims memory occupied by objects that are no longer in use (no longer referenced). This prevents memory leaks.
     - pymalloc: Python uses pymalloc, an

optimized memory allocator designed for small objects, which improves performance.

4. Immutability vs. Mutability:
   - Understanding immutability and mutability is crucial for memory management:
     - Immutable Objects: Objects like strings, tuples, and numbers are immutable. Once created, their values cannot be changed. If you appear to change an immutable object, you're actually creating a new object.
     - Mutable Objects: Objects like lists and dictionaries are mutable. Their values can be changed after they're created.
5. Memory Management Details:
   - Python's memory management is complex and involves: ◦ Reference counting: keeping track of how many references an object has. ◦ Garbage collection to handle reference cycles (situations where objects refer to each other, preventing their reference counts from reaching zero). ◦ The CPython implementation, which is the most common python implementation, has very specific ways of handling memory, and

those ways can change between python versions.

Q3.Python operators: arithmetic, comparison, logical, bitwise.

Python provides a variety of operators to perform different operations. Here's a breakdown of the main categories:

1. Arithmetic Operators:

- These operators perform mathematical calculations.
  - +: Addition (e.g., x + y) ◦ -: Subtraction (e.g., x - y) ◦ *: Multiplication (e.g., x * y)
  - /: Division (e.g., x / y)
  - %: Modulus (remainder) (e.g., x % y) ◦ **: Exponentiation (e.g., x ** y)
  - //: Floor division (integer division) (e.g., x // y)

2. Comparison Operators:

- These operators compare values and return a Boolean result (True or False). ◦ ==: Equal to (e.g., x == y) ◦ !=: Not equal to (e.g., x != y) ◦ >:

Greater than (e.g., x > y) ◦ <: Less than (e.g., x < y) ◦ >=: Greater than or equal to (e.g., x >= y) ◦ <=: Less than or equal to (e.g., x <= y)

3. Logical Operators:

- These operators combine or modify Boolean values. ◦ and: Returns True if both operands are True (e.g., x and y)
  - or: Returns True if at least one operand is True (e.g., x or y)
  - not: Returns the opposite of the operand's Boolean value (e.g., not x)

4. Bitwise Operators:

- These operators perform operations on individual bits of integers. ◦ &: Bitwise AND (e.g., x & y)
  - |: Bitwise OR (e.g., x | y)
  - ^: Bitwise XOR (exclusive OR) (e.g., x ^ y)
  - ~: Bitwise NOT (inverts the bits) (e.g., ~x) ◦ <<: Left shift (e.g., x << 2) ◦ >>: Right shift (e.g., x >> 2)

LAB:

Q1. Write a Python program to demonstrate the creation of variables and different data types.

```
def demonstrate_data_types():
"""Demonstrates the creation of variables and different data types in Python.""" # Integer (int) age = 30 print(f"Variable 'age' (int): {age}, type: {type(age)}") # Floating-point number (float) price = 19.99 print(f"Variable 'price' (float): {price}, type: {type(price)}") # String (str) name = "Alice" print(f"Variable 'name' (str): {name}, type: {type(name)}") # Boolean (bool) is_student = True print(f"Variable 'is_student' (bool): {is_student}, type: {type(is_student)}") # List (list) numbers = [1, 2, 3, 4, 5] print(f"Variable 'numbers' (list): {numbers}, type: {type(numbers)}") # Tuple (tuple) coordinates = (10, 20) print(f"Variable 'coordinates' (tuple): {coordinates}, type: {type(coordinates)}") # Dictionary (dict) person
```

= {"name": "Bob", "age": 25, "city": "New York"} print(f"Variable 'person' (dict): {person}, type: {type(person)}") # Set (set) unique_numbers = {1, 2, 3, 3, 4, 5} print(f"Variable 'unique_numbers' (set): {unique_numbers}, type: {type(unique_numbers)}") # NoneType (None) empty_variable = None print(f"Variable 'empty_variable' (None): {empty_variable}, type: {type(empty_variable)}") # Demonstrating type conversion (casting) string_number = "123" integer_number = int(string_number) #string to int print(f"Converted string '{string_number}' to integer: {integer_number}, type: {type(integer_number)}") float_age = float(age) #int to float print(f"Converted integer '{age}' to float: {float_age}, type: {type(float_age)}") if __name__ == "__main__": demonstrate_data_types()

Q2. Practical Example 1: How does the Python code structure work?

 # 1. Module-level code (definitions and initializations) # Global variables (less common, but possible) tax_rate = 0.08 # 8% sales tax # Function definition: calculate the total cost def calculate_total(items): """Calculates the total cost of items in a shopping cart."""" subtotal = 0 for item in items: subtotal += item["price"] * item["quantity"] #access dictionary values tax = subtotal * tax_rate total = subtotal + tax return total # Function definition: add an item to a list def add_item(cart, name, price, quantity): """Adds a item to the cart list""" item = {"name": name, "price": price, "quantity": quantity} cart.append(item) # 2. Main execution block (where the program starts) if __name__ == "__main__": # Initialize an empty shopping cart (a list of dictionaries) shopping_cart = [] # Add items to the cart

add_item(shopping_cart, "Laptop", 1200, 1)

add_item(shopping_cart, "Mouse", 25, 2)

add_item(shopping_cart, "Keyboard", 50, 1) # Calculate and print the total cost total_cost = calculate_total(shopping_cart) print(f"Total cost: ${total_cost:.2f}") #formatted output print("Items in cart:") for item in shopping_cart: print(f" {item['name']}: ${item['price']} x {item['quantity']}")

Q3. Practical Example 2: How to create variables in Python?

def variable_creation_examples():

"""Demonstrates various ways to create variables in Python.""" **# 1. Basic Assignment name = "John Doe" # String variable age = 30 # Integer variable height = 5.9 # Float variable is_student = True # Boolean variable print(f"Name: {name}, Type: {type(name)}") print(f"Age: {age}, Type: {type(age)}") print(f"Height: {height}, Type: {type(height)}")**

print(f"Is Student: {is_student}, Type: {type(is_student)}") # 2. Multiple Assignments x, y, z = 10, "Hello", 3.14 # Assign multiple values at once print(f"x: {x}, y: {y}, z: {z}") a = b = c = 50 # Assign the same value to multiple variables print(f"a: {a}, b: {b}, c: {c}") # 3. Variable Reassignment count = 5 print(f"Initial count: {count}") count = 10 # Reassign the variable print(f"Updated count: {count}") # 4. Dynamic Typing dynamic_var = 10 print(f"dynamic_var (int): {dynamic_var}, type: {type(dynamic_var)}") dynamic_var = "Python" # Reassign with a different type print(f"dynamic_var (str): {dynamic_var}, type: {type(dynamic_var)}") # 5. Using Variables in Expressions width = 10 length = 20 area = width * length print(f"Area: {area}") # 6. Variable Naming Conventions # - Use descriptive names. # - Use lowercase with underscores (snake_case). # - Avoid reserved

keywords (e.g., 'if', 'for', 'while'). # - Start with a letter or underscore. my_variable = 42 user_name = "Alice" _temp_value = 100 #leading underscore is allowed. #7. unpacking variables coordinates = (10, 20, 30) x_coord, y_coord, z_coord = coordinates print(f"coordinates: x={x_coord}, y={y_coord}, z={z_coord}") #8. unpacking lists data = ['apple', 5, True] fruit, quantity, available = data print(f'fruit: {fruit}, quantity: {quantity}, available: {available}') if __name__ == "__main__": variable_creation_examples()

Q3. Practical Example 3: How to take user input using the input() function.

def user_input_example(): """Demonstrates how to take user input in Python.""" # 1. Taking string input name = input("Enter your name: ") print(f"Hello, {name}!") # 2. Taking integer input try: age = int(input("Enter your age: ")) print(f"You are {age} years old.")

except ValueError: print("Invalid input. Please enter a valid integer for age.") # 3. Taking float input try: height = float(input("Enter your height (in meters): ")) print(f"Your height is {height:.2f} meters.") except ValueError: print("Invalid input. Please enter a valid number for height.") # 4. Taking multiple inputs (space-separated) try: num1, num2 = map(int, input("Enter two numbers separated by space: ").split()) sum_result = num1 + num2 print(f"The sum of {num1} and {num2} is: {sum_result}") except ValueError: print("Invalid input. Please enter two integers separated by a space.") # 5. Taking boolean-like input (converting string to boolean) is_student_input = input("Are you a student? (yes/no): ").lower() #.lower() for case insensitivity is_student = is_student_input == "yes" #boolean comparison. print(f"Student status: {is_student}") #6. using input with a

default value. city = input("Enter your city (default: New York): ") or "New York" #if input is empty, default is used. print(f"City: {city}") if __name__ == "__main__": user_input_example()

Q4. Practical Example 4: How to check the type of a variable dynamically using type().

def dynamic_type_checking(): """Demonstrates how to dynamically check variable types using type()."""" # Example variables with different types variable1 = 10 # Integer variable2 = "Hello" # String variable3 = 3.14 # Float variable4 = [1, 2, 3] # List variable5 = {"name": "Alice", "age": 30} # Dictionary variable6 = True #Boolean variable7 = None #NoneType # Function to check and print the type def check_and_print_type(variable, variable_name): """Checks the type of a variable and prints it."""" variable_type =

type(variable) print(f"Variable '{variable_name}' has type: {variable_type}") # You can use type() for conditional logic if variable_type is int: print(f" '{variable_name}' is an integer.") elif variable_type is str: print(f" '{variable_name}' is a string.") elif variable_type is list: print(f" '{variable_name}' is a list.") elif variable_type is dict: print(f" '{variable_name}' is a dictionary.") elif variable_type is float: print(f" '{variable_name}' is a float.") elif variable_type is bool: print(f" '{variable_name}' is a boolean.") elif variable_type is type(None): #or variable_type is NoneType print(f" '{variable_name}' is NoneType.") # Check and print types of the variables check_and_print_type(variable1, "variable1") check_and_print_type(variable2, "variable2") check_and_print_type(variable3, "variable3") check_and_print_type(variable4, "variable4")

```
check_and_print_type(variable5, "variable5")
check_and_print_type(variable6, "variable6")
check_and_print_type(variable7, "variable7")
# Example of type checking in a function def
process_variable(data): """Processes data
based on its type.""" if type(data) is int:
print(f"Processing integer: {data * 2}") elif
type(data) is str: print(f"Processing string:
{data.upper()}") else: print("Unsupported data
type.") process_variable(25)
process_variable("example")
process_variable([1,2,3]) if __name__ ==
"__main__": dynamic_type_checking()
```

4. Conditional Statements

Q1.Introduction to conditional statements: if, else, elif.

Conditional statements are fundamental to programming, as they allow your code to make decisions and execute different blocks of code based on whether certain conditions are true or false. Python uses the if, elif (else if), and else keywords to create these conditional statements. Here's a breakdown:

1. if Statement:

- The if statement is the most basic conditional statement. It evaluates a condition, and if the condition is true, it executes the code block that follows.
- Syntax: Python

```
if condition:
    # code to execute if condition is True
```

- Example: Python x = 10 if x > 5:

```
print("x is greater than 5")
```

2. else Statement:

- The else statement is used in conjunction with the if statement. It provides a block of code that will be executed if the if condition is false.
- Syntax: Python

```
if condition:
    # code to execute if condition is True else:

    # code to execute if condition is False
```

- Example: [1]

1. github.com

github.com

Python x = 3 if x > 5:

print("x is greater than 5") else:

print("x is not greater than 5")

3. elif Statement:

- The elif (else if) statement allows you to check multiple conditions in a sequence. It's used when you have more than two possible outcomes.
- Syntax:

Python

if condition1:

code to execute if condition1 is True elif

condition2:

code to execute if condition2 is True else:

code to execute if all conditions are False

- Example: Python x = 7 if x > 10: print("x is greater than 10") elif x > 5:

print("x is greater than 5, but not greater than 10") else:

print("x is 5 or less")

Q2. Nested if-else conditions.

Nested if-else statements occur when you place an if, elif, or else statement inside another if or else block. This allows for more complex decision-making processes in your code. Here's a breakdown:

Purpose:

- Nested if-else statements are used when you need to evaluate multiple conditions that depend on the outcome of previous conditions.
- They help create a hierarchical decision structure, where subsequent checks are performed only if certain prior conditions are met.

How it Works:

- The outer if statement is evaluated first.
- If the outer condition is true, the code block within that if statement is executed.
- If that code block contains another if statement (a nested if), that inner if statement is then evaluated.

- This process can continue, with if statements nested within if statements, creating deeper levels of decision-making.
- If the outer if statement is false, then the outer else statement, if one exists, will be executed. If that else statement contains further if statements, those will then be evaluated.

Example:

Python

```
x = 10

y = 5

 if x > 5:     print("x is greater than 5")

if y > 2:         print("and y is greater

than 2")     else:

        print("but y is not greater than 2") else:

   print("x is not greater than 5") In
```

this example:

- The outer if statement checks if x is greater than 5.
- If it is, the code inside that if block is executed.
- That code block contains another if statement that checks if y is greater than 2.

Key Considerations:

- Readability: Deeply nested if-else statements can become difficult to read and understand.

  It's important to use clear indentation and consider alternative approaches (like logical operators or functions) for complex conditions.
- Logic: Carefully plan your logic to ensure that your nested conditions behave as expected.
- Alternatives: In some cases, you can simplify complex nested if-else structures by using logical operators (and, or) to combine conditions.

  LAB:

  Q1. Practical Example 5: Write a Python program to find greater and less than a number using if_else.

  def compare_numbers(): """Compares a user-entered number with a predefined number using if-else."""" predefined_number = 50 # The number to compare against try: user_number = int(input("Enter a number: ")) if user_number > predefined_number: print(f"{user_number} is greater than {predefined_number}.") elif user_number < predefined_number: print(f"{user_number} is less than {predefined_number}.") else:

print(f"{user_number} is equal to {predefined_number}.") except ValueError: print("Invalid input. Please enter a valid integer.") if __name__ == "__main__": compare_numbers()

Q2. Practical Example 6: Write a Python program to check if a number is prime using if_else

def is_prime(number): """Checks if a number is prime using if-else.""" if number <= 1: return False # Numbers less than or equal to 1 are not prime if number <= 3: return True # 2 and 3 are prime if number % 2 == 0 or number % 3 == 0: return False # Numbers divisible by 2 or 3 are not prime i = 5 while i * i <= number: if number % i == 0 or number % (i + 2) == 0: return False # Numbers divisible by i or i+2 are not prime i += 6 return True # If no divisors are found, the number is prime def main(): """Gets user input and checks if the number is prime.""" try: num = int(input("Enter a number: ")) if is_prime(num): print(f"{num} is a prime number.") else: print(f"{num} is not a prime number.") except ValueError: print("Invalid input. Please enter an integer.") if __name__ == "__main__": main()

Q3. Practical Example 7: Write a Python program to calculate grades based on percentage using if-else ladder.

def calculate_grade(percentage): """Calculates the grade based on the given percentage. Args: percentage: The percentage score. Returns: The corresponding grade as a string. """ if percentage >= 90: return "A+" elif percentage >= 80: return "A" elif percentage >= 70: return "B" elif percentage >= 60: return "C" elif percentage >= 50: return "D" else: return "F" # Example usage: percentage = float(input("Enter the percentage: ")) if 0 <= percentage <= 100: #validate input grade = calculate_grade(percentage) print(f"The grade for {percentage}% is: {grade}") else: print("Invalid percentage. Please enter a percentage between 0 and 100.")

Q4. Practical Example 8: Write a Python program to check if a person is eligible to donate blood using a nested if.

def check_blood_donation_eligibility(age, weight, hemoglobin): """Checks if a person is eligible to donate blood. Args: age: The age of the person (in years). weight: The weight of the

person (in kilograms). hemoglobin: The hemoglobin level (in g/dL). Returns: A string indicating eligibility or ineligibility. """ if age >= 18: if weight >= 50: if hemoglobin >= 12.5: return "Eligible to donate blood." else: return "Ineligible: Hemoglobin level too low." else: return "Ineligible: Weight too low." else: return "Ineligible: Age too low." # Example usage: age = int(input("Enter your age: ")) weight = float(input("Enter your weight (in kg): ")) hemoglobin = float(input("Enter your hemoglobin level (in g/dL): ")) eligibility = check_blood_donation_eligibility(age, weight, hemoglobin) print(eligibility)

5. Looping (For, While)

Q1. Introduction to for and while loops.

Loops are essential control flow structures in programming that allow you to execute a block of code repeatedly. Python provides two primary loop types: for loops and while loops. Here's an introduction to each:

1. for Loops:

- Purpose:

- for loops are designed for iterating over a sequence (like a list, tuple, string, or range) or other iterable objects. ○ They execute a block of code for each item in the sequence.

- Syntax: Python for variable in sequence:

  # code to execute

- Example: Python fruits = ["apple", "banana", "cherry"] for fruit in fruits:

  print(fruit)

- Key Characteristics: ○ Ideal when you know in advance how many times you need to iterate. ○ Often used to process each element of a collection. ○ The range() function is very commonly used with for loops.

2. while Loops:

- Purpose: ○ while loops repeatedly execute a block of code as long as a specified condition is true.

  - They continue looping until the condition becomes false.

- Syntax: Python while condition:

  # code to execute

- Example: Python

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

- Key Characteristics: ◦ Useful when you don't know in advance how many times you need to iterate. ◦ Suitable for situations where the loop's termination depends on a dynamic condition. ◦ It is very important to make sure that the condition within a while loop will eventually become false. If it does not, the

  while loop will run indefinatly, creating what is known as an infinite loop.

Key Differences:

- Iteration Control:
  - for loops iterate over a predefined sequence.
  - while loops iterate based on a condition.
- Use Cases:

- for loops are best for iterating over collections. - while loops are best for repeating code until a condition is met.

Both for and while loops are fundamental tools in Python programming, and understanding their differences is crucial for writing efficient and effective code.

Q2. How loops work in Python.

Python loops are fundamental control flow structures that allow you to repeat a block of code multiple times. Python primarily uses two types of loops: for loops and while loops. Here's a breakdown of how they work:

1. for Loops:

- Iteration over Sequences:
  - for loops are designed to iterate over sequences like lists, tuples, strings, or ranges. - They execute a block of code for each element in the sequence.
- How it Works: - The for loop takes each item from the sequence, one at a time, and assigns it to a variable. - Then, it executes the code within the loop's block. - This process repeats until all items in the sequence have been processed.

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

* In this example, the loop iterates through the `fruits` list, and for each fruit, it prints the fruit's name.

- The range() Function:
  - The range() function is often used with for loops to generate a sequence of numbers. ◦ For example, range(5) generates the sequence 0, 1, 2, 3, 4.

2. while Loops:

- Condition-Based Iteration: ◦ while loops repeat a block of code as long as a specified condition is true. ◦ They continue looping until the condition becomes false.
- How it Works: ◦ The while loop checks the condition before each iteration. ◦ If the condition is true, it executes the code within the loop's block. ◦ If the condition is false, the loop terminates.
- Example: Python

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

* In this example, the loop continues as long as `count` is less than 5.

* In each iteration, it prints the value of `count` and increments it by 1.

- Important Note:
  - It's crucial to ensure that the condition in a while loop will eventually become false. Otherwise, you'll create an infinite loop.

Key Concepts:

- Iteration: The process of repeating a block of code.
- Condition: A Boolean expression that determines whether a loop should continue.
- Loop Body: The block of code that is executed repeatedly.

Q3. Using loops with collections (lists, tuples, etc.).

Using loops with collections like lists, tuples, and dictionaries is a very common and powerful practice in

Python. Here's a breakdown of how it works and some common use cases:

1. for Loops and Collections:

- Iterating Through Elements:
  - The for loop is the primary tool for iterating through the elements of a collection. ◦ It allows you to access each element in the collection one by one and perform operations on it.
- Lists and Tuples: ◦ Lists and tuples are ordered sequences, so for loops iterate through their elements in the order they appear. ◦ Example:

Python

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


numbers = (1, 2, 3, 4, 5)
for number in numbers:
print(number * 2)
```

Dictionaries:

- Dictionaries are collections of key-value pairs. ◦ When you iterate through a dictionary with a for loop, you iterate through its keys by

default. ◦ To iterate through key-value pairs, you can use the .items() method. ◦ Example: Python student = {"name": "Alice", "age": 20, "major": "Computer Science"} for key, value in student.items():

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

2. Common Use Cases:

- Processing Data: ◦ Loops are used to process data stored in collections, such as calculating sums, finding maximums, or filtering elements.
- Transforming Data: ◦ You can use loops to transform data by creating new collections based on the elements of existing ones.
- Searching and Filtering:
  - Loops allow you to search for specific elements in a collection or filter elements based on certain criteria.
- Performing Actions on Each Element: ◦ Loops are perfect for when you need to perform the same action on every item within a collection.

3. Additional Techniques:

- enumerate(): ◦ The enumerate() function is useful when you need to access both the index and the value of each element in a collection. ◦ Example: Python fruits = ["apple", "banana", "cherry"] for index, fruit in enumerate(fruits): print(f"Index {index}: {fruit}")

LAB:

Q1. Practical Example 1: Write a Python program to print each fruit in a list using a simple for loop. List1 = ['apple', 'banana', 'mango']

List1 = ['apple', 'banana', 'mango'] for fruit in List1: print(fruit)

Q2. Practical Example 2: Write a Python program to find the length of each string in List1

List1 = ['apple', 'banana', 'mango'] for fruit in List1: length = len(fruit) print(f"The length of '{fruit}' is: {length}")

Q3. Practical Example 3: Write a Python program to find a specific string in the list using a simple for loop and if condition.

List1 = ['apple', 'banana', 'mango', 'orange', 'grape'] search_string = input("Enter the string to search: ") found = False # Initialize a flag to track if

the string is found for fruit in List1: if fruit == search_string: print(f"'{search_string}' found in the list!") found = True break # Exit the loop once the string is found if not found: print(f"'{search_string}' not found in the list.")

Q4. Practical Example 4: Print this pattern using nested for loop: markdown Copy code * ** * **** *******

rows = 5 # Number of rows in the pattern for i in range(1, rows + 1): # Outer loop for rows for j in range(i): # Inner loop for columns (stars in each row) print("*", end="") # Print a star without a newline print() # Move to the next line after each row

6. Generators and Iterators

Q1.Understanding how generators work in Python

Generators in Python are a powerful tool for creating iterators in a memory-efficient way. They allow you to define a function that behaves like an iterator, yielding values one at a time, rather than storing the entire sequence in memory. Here's a breakdown of how they work:

1. Iterators vs. Generators:

- Iterators: ◦ Iterators are objects that allow you to traverse through a sequence of data. ◦ They implement the __iter__() and __next__() methods. ◦ They produce values on demand.
- Generators: ◦ Generators are a simpler way to create iterators. ◦ They use the yield keyword to produce values. ◦ They automatically handle the __iter__() and __next__() methods.

2. The yield Keyword:

- The yield keyword is the key to generators.
- When a yield statement is encountered, the generator function's state is saved, and the yielded value is returned.
- The next time the generator [1] is called, it resumes execution from where it left off.

1. media.licdn.com

media.licdn.com

- This "pausing and resuming" behavior is what makes generators memory-efficient.

3. How Generators Work:

- Function Definition:
  - A generator is defined like a regular function, but it contains one or more yield statements.
- Generator Object:
  - When you call a generator function, it doesn't execute the code immediately. Instead, it returns a generator object.
- Iteration:
  - You can iterate over the generator object using a for loop or the next() function.
  - Each time you request a value, the generator executes until it encounters a yield statement.
  - The yielded value is returned, and the generator's state is saved.
  - Once the generator has no more values to yield, it raises a StopIteration exception.
- Memory Efficiency:
  - Generators produce values on demand, so they don't need to store the entire sequence in memory.
  - This is especially useful for working with large datasets or infinite sequences.

4. Example: Python def

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

# Creating a generator object
my_generator = count_up_to(5)

# Iterating over the generator for
num in my_generator:
    print(num)

#Or using next() my_second_generator =
count_up_to(3)
print(next(my_second_generator))
print(next(my_second_generator))
print(next(my_second_generator))
```

5. Use Cases:

- Large Datasets: Generators are ideal for processing large datasets that don't fit into memory.
- Infinite Sequences: Generators can represent infinite sequences, such as streams of data.
- Lazy Evaluation: Generators allow for lazy evaluation, where values are computed only when needed.
- Improved Readability: Generators can make code more readable and concise, especially for complex iteration logic.

Q2. Difference between yield and return.

The yield and return keywords in Python serve distinct purposes, particularly concerning how functions execute and return values. Here's a breakdown of their key differences:

1. Function Behavior:

- return: ◦ Terminates the function's execution immediately. ◦ Sends a specified value back to the caller. ◦ The function's state is not preserved.
- yield:
  - ◦ Pauses the function's execution. ◦ Sends a value back to the caller. ◦ The function's state

is preserved, allowing it to resume execution from where it left off when called again.

2. Function Type:
   - return:

- Used in regular functions.

- yield: Used in generator functions. When a python function contains the keyword yield, that function becomes a generator.

3. Return Values:
   - return:
     - Returns a single value or a tuple of values.
   - yield:
     - Yields a series of values, one at a time, creating an iterator.
4. Memory Usage:
   - return:
     - May require storing all return values in memory at once.
   - yield: Generates values on demand, resulting in more memory-efficient processing, especially for large datasets.
5. Iteration:
   - return:

     Does not inherently support iteration.

- yield:
  - Enables iteration by creating a generator object.

Q3. Understanding iterators and creating custom iterators.

Iterators are a fundamental concept in Python, enabling you to traverse through elements of a collection or sequence in a controlled and efficient manner. Here's a comprehensive overview:

1. What are Iterators?

- Iterators are objects that allow you to access elements of a container (like a list, tuple, or dictionary) sequentially.
- They provide a way to access elements without needing to load the entire collection into memory at once.
- They implement the iterator protocol, which consists of two methods:
  - __iter__(): Returns the iterator object itself.

    __next__(): Returns the next element [1] in the sequence. If there are no more elements, it raises a StopIteration exception.

1. www.geekster.in

-

www.geekster.in

2. How Iterators Work:

- When you create an iterator from a collection, the __iter__() method is called.
- To retrieve the next element, you call the __next__() method.
- The iterator keeps track of its current position in the sequence.
- When there are no more elements, the __next__() method raises a StopIteration exception, signaling the end of the iteration.
- Python's for loops internally use iterators to traverse collections.

3. Built-in Iterators:

- Python provides built-in iterators for various data types, including lists, tuples, dictionaries, and strings.
- You can create an iterator from these collections using the iter() function. ○ Example: Python

my_list = [1, 2, 3] my_iterator = iter(my_list) print(next(my_iterator))
# Output: 1 print(next(my_iterator)) # Output: 2 print(next(my_iterator)) # Output: 3
# next(my_iterator) # would raise StopIteration.

4. Creating Custom Iterators:

- You can create your own iterators by defining a class that implements the iterator protocol (__iter__() and __next__()).
- Example: Python class MyIterator:

def __init__(self, max_value): self.max_value = max_value self.current_value = 0

```
        def __iter__(self):
return self


        def __next__(self):
            if self.current_value < self.max_value:
                result = self.current_value
self.current_value += 1
return result             else:
                raise StopIteration


    my_iterator = MyIterator(3)
for value in my_iterator:
        print(value)
```

Explanation of the Custom Iterator:

- The MyIterator class initializes with a max_value.
- The __iter__() method returns the iterator object itself.
- The __next__() method checks if the current_value is less than the max_value.
- If it is, it returns the current_value and increments it.

- If it's not, it raises a StopIteration exception.

5. Use Cases for Custom Iterators:
   - Generating complex sequences: You can create iterators that generate sequences based on custom logic.
   - Reading large files: Iterators can be used to read large files line by line, without loading the entire file into memory.
   - Working with infinite sequences: Iterators can represent infinite sequences, such as streams of data.
   - Custom data structures: when creating your own data structures, you may need to create custom iterators so that users can loop over your data structures.

LAB:

Q1. Write a generator function that generates the first 10 even numbers.

def generate_even_numbers(limit=10): """Generates the first 'limit' even numbers.""" count = 0 num = 0 while count < limit: yield num num += 2 count += 1 # Example usage: even_numbers = generate_even_numbers() for number in

even_numbers: print(number) #or to get the first 10 even numbers as a list. even_list = list(generate_even_numbers()) print(even_list)

Q2. Write a Python program that uses a custom iterator to iterate over a list of integers.

class IntegerIterator: """A custom iterator for iterating over a list of integers."""" def __init__(self, data): self.data = data self.index = 0 def __iter__(self): return self def __next__(self): if self.index < len(self.data): result = self.data[self.index] self.index += 1 return result else: raise StopIteration # Example usage: my_list = [10, 20, 30, 40, 50] my_iterator = IntegerIterator(my_list) for item in my_iterator: print(item) #Alternative usage showing next() my_iterator2 = IntegerIterator(my_list) try: while True: print(next(my_iterator2)) except StopIteration: pass #Iterator is exhausted.

7. Functions and Methods

Q1. Defining and calling functions in Python.

Functions are fundamental building blocks in Python, allowing you to organize code into reusable blocks. Here's how you define and call functions:

1. Defining Functions:

- You define a function using the def keyword, followed by the function name, parentheses (), and a colon :.
- Inside the parentheses, you can specify parameters (inputs) that the function accepts.
- The code block within the function is indented.
- You can optionally use the return keyword to send a value back to the caller.

Syntax: Python def function_name(parameter1, parameter2, ...):

Function body (code to execute)

Optional: return value return value

Example: Python def greet(name):

"""This function greets the person passed in as a parameter.""" print(f"Hello, {name}!")

def add(x, y):

"""This function adds two numbers and returns the result.""" result = x + y return result

Explanation:

- def greet(name): defines a function named greet that takes one parameter, name.
- def add(x, y): defines a function named add that takes two parameters, x and y.
- The lines within the indented code block are the function's body.
- The return statement in the add function sends the calculated result back to the caller.
- The docstrings (the text within triple quotes """ """) are used to document the function's purpose.

2. Calling Functions:

To execute a function, you "call" it by using its name followed by parentheses ().

- If the function has parameters, you provide the corresponding arguments (values) within the parentheses.
- If the function returns a value, you can store it in a variable.

Example: Python greet("Alice") # Calls the greet function with the argument "Alice"

sum_result = add(5, 3) # Calls the add function with arguments 5 and 3 print(sum_result) # Output: 8 Explanation:

- greet("Alice") calls the greet function and passes the string "Alice" as the argument for the name parameter.
- sum_result = add(5, 3) calls the add function, passes 5 and 3 as arguments, and stores the returned value (8) in the sum_result variable.

Key Concepts:

Parameters: Variables defined in the function's definition that receive values when the function is called.

Arguments: Actual values passed to the function when it is called.

- Return Value: The value that a function sends back to the caller. If a return statement is not used, python will return None.
- Scope: Variables defined within a function have local scope, meaning they are only accessible within the function.
- Docstrings: Documentation strings used to describe the function's purpose, parameters, and return value.

Q2. Function arguments (positional, keyword, default).

Python functions offer flexibility in how you pass arguments, providing several ways to define and use them. Here's a breakdown of positional, keyword, and default arguments:

1. Positional Arguments:

These are the most common type of arguments.

Their order matters. The arguments are matched to the parameters based on their position in the function call.

Example: Python def

```
describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type}.")
print(f"My {animal_type}'s name is {pet_name}.")


describe_pet("hamster", "Harry")  # "hamster" is assigned to animal_type, "Harry" to pet_name
describe_pet("Harry", "hamster") # "Harry" is assigned to animal_type, "hamster" to pet_name.
```

- In the first call, "hamster" is the first positional argument, so it's assigned to the animal_type parameter, and "Harry" is assigned to pet_name.

- In the second call, the arguments positions are switched, and thus, so are the parameter assignments.

2. Keyword Arguments:

   Keyword arguments allow you to pass arguments by explicitly naming the parameters.

   The order of keyword arguments doesn't matter.

- Example: Python def describe_pet(animal_type, pet_name):

  print(f"I have a {animal_type}.")
print(f"My {animal_type}'s name is {pet_name}.")

describe_pet(pet_name="Harry", animal_type="hamster")

- Here, we're using the parameter names (pet_name, animal_type) to specify which argument corresponds to which parameter.

3. Default Arguments:

- Default arguments allow you to specify a default value for a parameter.
- If a value is not provided for that parameter when the function is called, the default value is used.

Default arguments must come after positional arguments in the function definition.

Example: Python def describe_pet(pet_name, animal_type="dog"):

print(f"I have a {animal_type}.") print(f"My {animal_type}'s name is {pet_name}.")

describe_pet("Willie") # animal_type defaults to "dog" describe_pet("Sparky", "cat") # animal_type is explicitly set to "cat"

- In the first call, we only provide the pet_name, so animal_type defaults to "dog".
- In the second call, we provide both arguments, so the default value is overridden.
- Important: mutable default arguments can lead to unexpected behavior. It is usually best to avoid using mutable objects as default arguments.

Combining Argument Types:

You can combine positional, keyword, and default arguments in a single function call.

However, positional arguments must come first, followed by keyword arguments.

- Example: Python def my_function(a, b, c=10): print(f"a: {a}, b: {b}, c: {c}")

my_function(1, 2) # a=1, b=2, c=10 (default)

my_function(1, 2, 3) # a=1, b=2, c=3

my_function(1, b=2) # a = 1, b = 2, c = 10 Q3.

Scope of variables in Python.

In Python, the scope of a variable determines where in your code that variable can be accessed. Understanding variable scope is crucial for writing clean and error-free code. Python primarily uses four types of variable scope:

1. Local Scope:

- Variables defined inside a function have local scope.

- They are only accessible within that function.

  When the function finishes executing, the local variables are destroyed.

  Each function call creates a new local scope.

2. Global Scope:

- Variables defined outside of any function or class have global scope.
- They can be accessed from anywhere within the program, [1] including inside functions. [2]

1. github.com

github.com

2. github.com

-
-

github.com

However, to modify a global variable from within a function, you must use the global keyword.

3. Enclosing (Nonlocal) Scope:

- This scope applies to nested functions (functions within functions).
- If a variable is defined in an outer function and accessed in an inner function, it has an enclosing or nonlocal scope.
- To modify a variable in the enclosing scope from within the inner function, you must use the nonlocal keyword.

4. Built-in Scope:

- This scope contains built-in names, such as functions like print() and len(), and exceptions like ValueError.
- These names are always available throughout your program.

Key Points:

- LEGB Rule: Python follows the LEGB rule to determine the scope of a variable:
    - Local
    - Enclosing

- Global
- Built-in

- Variable Shadowing: If a local variable has the same name as a global variable, the local variable takes precedence within the function.
- global Keyword:
  - Used to declare that a variable within a function refers to a global variable.
  - Allows you to modify global variables from within a function.
- nonlocal Keyword:
  - Used to declare that a variable within a nested function refers to a variable in the enclosing function's scope.
  - Allows you to modify variables in the enclosing scope.

Q4. Built-in methods for strings, lists, etc.

Python provides a rich set of built-in methods for working with various data types, including strings, lists, and dictionaries. Here's a look at some of the most commonly used methods:

Strings:

- upper():

-

Converts all characters in a string to uppercase.

lower(): ○ Converts all characters in a string to lowercase.

- strip(): ○ Removes leading and trailing whitespace from a string.
- split():
  - ○ Splits a string into a list of substrings based on a delimiter.
- join(): ○ Concatenates elements of an iterable (like a list) into a single string, using a specified separator.
- find(): ○ Returns the index of the first occurrence of a substring within a string.
- replace(): ○ Replaces occurrences of a substring with another substring.
- startswith():

-

returns true if the string starts with the given value.

endswith(): returns true if the string ends with the given value.

Lists:

- append():
  - Adds an element to the end of a list.
- insert(): Inserts an element at a specified position in a list.
- remove(): Removes the first occurrence of a specified element from a list.
- pop(): Removes and returns the element at a specified position (or the last element if no index is specified).
- sort(): Sorts the elements of a list in ascending order.
- reverse():

  Reverses the order of the elements in a list.

-

- count(): returns the number of elements with the specified value.
- extend(): Add the elements of a list (or any iterable), to the end of the current list.
- clear(): Removes all the elements from the list.

Dictionaries:

- keys():
  - Returns a view object that displays a list of all the keys in the dictionary.
- values():
  - Returns a view object that displays a list of all the values in the dictionary.
- items():
  - Returns a view object that displays a list of all the key-value pairs in the dictionary.
- get():

  Returns the value associated with a specified key.

- update(): ○ Updates the dictionary with the key-value pairs from another dictionary or iterable.
- pop(): ○ Removes the element with the specified key.
- clear(): ○ Removes all the elements from the dictionary.

LAB:

Q1. Practical Example: 1) Write a Python program to print "Hello" using a string

 # Method 1: Direct printing print("Hello") # Method 2: Storing in a variable and printing greeting = "Hello" print(greeting)

Q2. Practical Example: 2) Write a Python program to allocate a string to a variable and print it.

 # Allocate a string to a variable my_string = "This is a sample string." # Print the variable print(my_string) # You can also use f-strings for formatted output: name = "Alice" message = f"Hello, {name}!" print(message) # Or concatenate strings: part1 = "Python" part2 = " is fun!"

o

combined_string = part1 + part2
print(combined_string)

Q3. Practical Example: 3) Write a Python program to print a string using triple quotes.

 # Method 1: Single triple quotes multi_line_string = '''This is a string that spans multiple lines. It can include 'single' and "double" quotes.'''
print(multi_line_string) # Method 2: Double triple quotes another_multi_line_string = """This is another multi-line string, also with 'single' and "double" quotes."""
print(another_multi_line_string) # Method 3: Assigning to a variable, then printing. long_string = """ This is a very long string, that can be spread across multiple lines for better readability. """
print(long_string)

Q4. Practical Example: 4) Write a Python program to access the first character of a string using index value.

my_string = "Python" # Access the first character using index 0 first_character = my_string[0] # Print

o

the first character print(first_character) # Output: P #Accessing the first character directly in the print statement. print(my_string[0]) #output: P #Demonstrating a string with spaces. spaced_string = "Hello World" print(spaced_string[0]) #output: H

Q5. Practical Example: 5) Write a Python program to access the string from the second position onwards using slicing.

my_string = "Programming" # Access the string from the second position onwards (index 1 to the end) substring = my_string[1:] # Print the substring print(substring) # Output: rogramming #Another example with a different string. another_string = "Hello, World!" substring2 = another_string[1:] print(substring2) #output: ello, World! #Example showing slicing from a specific index to another. substring3 = my_string[3:7] print(substring3) #output: gram

Q6. Practical Example: 6) Write a Python program to access a string up to the fifth character

o

my_string = "Programming" # Access the string up to the fifth character (index 0 to 4) substring = my_string[:5] # Print the substring print(substring) # Output: Progr # Example with a shorter string short_string = "Hello" substring2 = short_string[:5] #or short_string[:] or short_string[0:5]. print(substring2) #Output: Hello #Example demonstrating negative slicing. example_string = "abcdefg" substring3 = example_string[:-2] #everything except the last two characters. print(substring3) #output: abcde

Q7. Practical Example: 7) Write a Python program to print the substring between index values 1 and 4

my_string = "ExampleString" # Access the substring between index 1 (inclusive) and 4 (exclusive) substring = my_string[1:4] # Print the substring print(substring) # Output: xam #Another example. another_string = "abcdefg" substring2 = another_string[1:4] print(substring2) #output: bcd #Example demonstrating that exceeding the end of the string does not cause an error. test_string =

o

"abc" substring3 = test_string[1:10] #index 10 does not exist. print(substring3) #output: bc

Q8. Practical Example: 8) Write a Python program to print a string from the last character.

my_string = "Python" # Access the last character using negative index -1 last_character = my_string[-1] # Print the last character print(last_character) # Output: n #Another way to print the last character directly. print(my_string[-1]) #output: n #Example with a longer string long_string = "Programming Language" print(long_string[-1]) #output: e #Example showing how to print the last 3 characters. last_three = long_string[-3:] print(last_three) #output: age

Q9. Practical Example: 9) Write a Python program to print every alternate character from the string starting from index 1.

my_string = "Programming" # Print every alternate character starting from index 1 alternate_chars = my_string[1::2] # Print the result print(alternate_chars) # Output: rgamn # Another

-

example: another_string = "abcdefghijklm" alternate_chars2 = another_string[1::2] print(alternate_chars2) #output: bdfhjlm # Example with a string that has spaces. spaced_string = "Hello World" alternate_chars3 = spaced_string[1::2] print(alternate_chars3) #output: el ol

8. Control Statements (Break, Continue, Pass)

Q1. Understanding the role of break, continue, and pass in Python loops break, continue, and pass are control flow statements used within Python loops to modify their behavior. Here's a breakdown of their roles:

1. break Statement:

- Purpose:
  - The break statement immediately terminates the loop (either for or while) in which it's encountered.
    - It transfers control to the statement immediately following the loop.

- Use Cases: ○ To exit a loop prematurely when a specific condition is met. ○ To stop an infinite loop when a desired result is found.
- Example: Python

```
for i in range(10):
if i == 5:
    break  # Exit the loop when i is 5
print(i)
```

#Output: 0 1 2 3 4 2.

continue Statement:

- Purpose: ○ The continue statement skips the rest of the current iteration of the loop. ○ It transfers control to the beginning of the next iteration.
- Use Cases:

- To skip certain iterations of a loop based on a condition.
- To avoid executing code that's not needed for specific loop iterations.

- Example: Python

```
for i in range(10):
if i % 2 == 0:
    continue  # Skip even numbers
print(i)
#Output: 1 3 5 7 9
```

3. pass Statement:

- Purpose:
  - The pass statement is a null operation. It does nothing.
  - It's used as a placeholder where a statement is syntactically required but no action needs to be taken.
- Use Cases:
  - To create empty functions or classes.
  - To create placeholders for code that will be implemented later.

  As a placeholder in conditional statements.
- Example: Python

```
def my_function():
    pass  # Placeholder for future implementation
```

```
if
True:
    pass #placeholder.

for i in range(5):          if i == 2:
pass #I will add code here later.
    else:
        print(i)
```

Key Differences:

- break terminates the entire loop.
- continue skips the current iteration and proceeds to the next.
- pass does nothing and is used as a placeholder.

LAB:

Q1. Practical Example: 1) Write a Python program to skip 'banana' in a list using the continue statement. List1 = ['apple', 'banana', 'mango']

List1 = ['apple', 'banana', 'mango'] for fruit in List1: if fruit == 'banana': continue # Skip the rest of the loop for 'banana' print(fruit)

-

Q2. Practical Example: 2) Write a Python program to stop the loop once 'banana' is found using the break statement.

List1 = ['apple', 'banana', 'mango', 'orange'] for fruit in List1: if fruit == 'banana': break # Exit the loop when 'banana' is found print(fruit)

9. String Manipulation

Q1. Understanding how to access and manipulate strings.

Strings in Python are sequences of characters and are fundamental for working with text data. Here's a breakdown of how to access and manipulate strings:

1. Accessing Characters:

- Indexing: You can access individual characters in a string using their index, starting from 0 for the first character. - Example: Python my_string = "Python" print(my_string[0]) # Output: P print(my_string[1]) # Output: y print(my_string[-1]) # Output: n, negative indexes count from the end.
- Slicing: You can extract a portion of a string using slicing. - Example:

Python my_string = "Python" print(my_string[1:4])

Output: yth print(my_string[:3]) # Output: Pyt

print(my_string[2:]) # Output: thon

print(my_string[:]) # Output: Python, creates a

copy of the string.

2. String Manipulation:

- Concatenation: You can combine strings using the + operator.
  - Example: Python

str1 = "Hello" str2 = "World" result =

str1 + ", " + str2 print(result) #

Output: Hello, World

- String Methods: Python provides a rich set of built-in string methods:
  - upper(): Converts the string to uppercase.
    - Example: "hello".upper() returns "HELLO".
  - lower(): Converts the string to lowercase.
    - Example: "HELLO".lower() returns "hello".
  - strip(): Removes leading and trailing whitespace.

- Example: " hello ".strip() returns "hello".
- split(): Splits the string into a list of substrings.
  - Example: "hello,world".split(",") returns ['hello', 'world'].
- join(): Joins a list of strings into a single string.
  - Example: "-".join(['hello', 'world']) returns "hello-world".
- find(): Finds the index of a substring.
  - Example: "hello".find("l") returns 2.
- replace(): Replaces a substring with another substring.
  - Example: "hello".replace("l", "x") returns "hexxo".
- startswith(): Checks if a string starts with a given prefix.
  - Example: "hello".startswith("he") returns True.
- endswith(): Checks if a string ends with a given suffix.
  - Example: "hello".endswith("lo") returns True.

3. String Formatting:

- f-strings (formatted string literals): A convenient way to embed expressions within string literals. ◦ Example: Python name = "Alice" age = 30 print(f"My name is {name} and I am {age} years old.")
- format() method: A versatile method for formatting strings. ◦ Example: Python name = "Bob" age = 25

print("My name is {} and I am {} years old.".format(name, age))

4. Immutability:

- Strings in Python are immutable, meaning you cannot change them directly. When you perform a string manipulation, you are creating a new string. ◦ Example: Python my_string = "Hello" my_string = my_string.replace("H", "J") # my_string now refers to a new string. print(my_string) #Output: Jello

Q2.Basic operations: concatenation, repetition,string methods(upper(), lower(), etc.)

Let's delve into the basic operations and methods for working with strings in Python:

1. Concatenation:

- Purpose: Joining two or more strings together.

- Operator: The + operator.
- Example:

Python

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


str3 = "Python"      num = "3"
result2 = str3 + num      print(result2)
#Output: Python3
```

- Important: You can only concatenate strings with other strings. If you try to concatenate a string with a number, you'll get a TypeError. You will need to convert the number into a string first.

2. Repetition:

- Purpose: Repeating a string a specified number of times.
- Operator: The * operator.
- Example: Python

```
str1 = "abc"
```

result = str1 * 3 print(result) #

Output: abcabcabc

3. String Methods:

- upper(): ◦ Converts all characters in a string to uppercase.
    - ◦ Example:

```
Python
str1 = "hello"          result = str1.upper()
print(result)  # Output: HELLO
```

- lower(): ◦ Converts all characters in a string to lowercase.
    - ◦ Example:

```
Python          str1 =
"WORLD"
result = str1.lower()
print(result)  #
Output: world
```

strip():

- Removes leading and trailing whitespace from a string.
  - Example:

```
Python
str1 = "  example  "
result = str1.strip()
print(result)  # Output: example
```

- lstrip():
  - Removes leading whitespace from a string.
  - Example:

```
Python
str1 = "  example  "
result = str1.lstrip()
print(result) #Output: example
```

- rstrip():
  - Removes trailing whitespace from a string.
  - Example:

```
Python
str1 = "  example  "
result = str1.rstrip()
print(result) #Output:   example
```

- split():
    - Splits a string into a list of substrings based on a delimiter (default is whitespace).
    - Example:

```
Python          str1 = "apple,banana,cherry"          result = str1.split(",")
print(result)  # Output: ['apple', 'banana', 'cherry']
```

- join():
    - Joins a list of strings into a single string using a specified separator.
    - Example:

```
Python
```

- my_list = ["apple", "banana", "cherry"]
result = "-".join(my_list) print(result) # Output: apple-banana-cherry
- find():
    - Returns the index of the first occurrence of a substring. Returns -1 if not found.
        - Example:

```
Python
str1 = "hello world" index = str1.find("world")
print(index) # Output: 6
```

- replace():
    - Replaces occurrences of a substring with another substring.
        - Example:

```
Python
str1 = "hello world" result =
str1.replace("world", "Python")
print(result) # Output: hello Python
```

- startswith():
    - Returns True if the string starts with the specified value, otherwise False.

- Example:

Python

```
str1 = "hello world"
result = str1.startswith("hello")
print(result) #Output: True
```

endswith():

- Returns True if the string ends with the specified value, otherwise False.
- Example:

Python

```
str1 = "hello world"
result = str1.endswith("world")
print(result) #Output: True
```

Q3. String slicing.

String slicing in Python is a powerful technique that allows you to extract portions of a string with ease. It's essential for various text manipulation tasks. Here's a breakdown of how it works:

-

Basic Concepts:

- Indexing: - Strings in Python are ordered sequences

  of characters. Each character has an index, starting from 0 for the first character.

  You can also use negative indices to access characters from the end of the string, where -1 represents the last character. [1]

1. blog.51cto.com

blog.51cto.com

- Slicing Syntax: - The general syntax for string slicing is string[start:stop:step]. - start: The index where the slice begins (inclusive). If omitted, it defaults to 0. - stop: The index where the slice ends (exclusive). If omitted, it defaults to the end

of the string. ○ step: The step size or increment. If omitted, it defaults to 1.

How String Slicing Works:

- Extracting Substrings:

-

  By specifying the start and stop indices, you can extract a substring from the original string. - For example, string[2:5] extracts the characters from index 2 up to (but not including) index 5.

- Omitting Indices:

  - If you omit the start index, the slice starts from the beginning of the string. - If you omit the stop index, the slice goes

    to the end of the string. - If you omit both, you get a copy of the entire string.

- Step Size: - The step parameter allows you to extract

  characters at specific intervals. - For example, string[::2] extracts every other character from the string. - A negative step value will reverse the string.

Key Uses:

- Parsing Data:

  Extracting specific pieces of information from text data. - Text Manipulation: - Modifying and rearranging strings.

- Data Validation: ○ Checking if strings conform to certain patterns.

LAB:

Q1. Write a Python program to demonstrate string slicing.

def demonstrate_string_slicing(my_string): """Demonstrates various string slicing operations."""" print(f"Original String: '{my_string}'") # Basic slicing: start and end indices print(f"Slice [2:7]: '{my_string[2:7]}'") # Characters from index 2 up to (but not including) 7 # Slicing from the beginning print(f"Slice [:5]: '{my_string[:5]}'") # Characters from the beginning up to (but not including) 5 # Slicing to the end print(f"Slice [5:]: '{my_string[5:]}'") # Characters from index 5 to the end # Slicing with a step print(f"Slice [::2]: '{my_string[::2]}'") # Every second character print(f"Slice [1::3]: '{my_string[1::3]}'") #Every third character starting at index 1. # Negative indexing (from the end) print(f"Slice [-3:]: '{my_string[-3:]}'") # Last three characters print(f"Slice [:-3]: '{my_string[:-3]}'") #Everything except the last three characters. print(f"Slice [-5:-2]:

o

'{my_string[-5:-2]}'") #Characters starting from fifth to last, up to second to last. # Reverse the string print(f"Reversed string: '{my_string[::-1]}'") #Slicing with negative step. print(f"Slice [5:1:-1]: '{my_string[5:1:-1]}'") #Reverse characters from index 5 to index 2. # Example usage: my_string = "Hello, Python!"
demonstrate_string_slicing(my_string)
my_string2 = "abcdefghijklmnopqrstuvwxyz"
demonstrate_string_slicing(my_string2)

Q2. Write a Python program that manipulates and prints strings using various string methods.

def string_manipulation_demo(text):
"""Demonstrates various string manipulation methods.""" **print(f"Original text: '{text}'") # Case manipulation print(f"Uppercase: '{text.upper()}'") print(f"Lowercase: '{text.lower()}'") print(f"Title case: '{text.title()}'") print(f"Swap case: '{text.swapcase()}'") # Removing whitespace text_with_spaces = " Hello, World! "**
print(f"Text with spaces: '{text_with_spaces}'") print(f"Stripped text:

'{text_with_spaces.strip()}'") print(f"Left stripped text: '{text_with_spaces.lstrip()}'") print(f"Right stripped text: '{text_with_spaces.rstrip()}'") # Finding and replacing print(f"Find 'World': {text.find('World')}") print(f"Replace 'World' with 'Python': '{text.replace('World', 'Python')}'") # Splitting and joining words = text.split() print(f"Split into words: {words}") joined_text = "-".join(words) print(f"Joined with '-': '{joined_text}'") # Checking string properties print(f"Is alphanumeric: {text.isalnum()}") print(f"Is alphabetic: {text.isalpha()}") print(f"Is digit: {text.isdigit()}") print(f"Is lowercase: {text.islower()}") print(f"Is uppercase: {text.isupper()}") print(f"Starts with 'Hello': {text.startswith('Hello')}") print(f"Ends with '!': {text.endswith('!')}") #Formatting strings. formatted_string = "The number is {:03d}".format(5) print(f"Formatted String: {formatted_string}") #f-strings name = "Alice" age = 30 f_string = f"My name is {name} and I am {age} years old." print(f_string) # Example usage: my_string = "Hello, World!"

- **string_manipulation_demo(my_string)**
  my_spaced_string = " This is a test. "
  string_manipulation_demo(my_spaced_string)
  my_test_string = "1234"
  string_manipulation_demo(my_test_string)
  my_test_string2 = "Hello123World"
  string_manipulation_demo(my_test_string2)

10. Advanced Python (map(), reduce(), filter(), Closures and Decorators)

Q1. How functional programming works in Python.

While Python is primarily known as a multiparadigm language (supporting object-oriented, procedural, and functional programming), it provides excellent tools for incorporating functional programming principles. Here's how functional programming works in Python:

Core Concepts:

- Pure Functions: These functions produce the same output for the same input and have no side effects (they don't modify external state). This makes them predictable and easier to test.
- Immutability: Functional programming favors immutable data structures, meaning data cannot be changed after it's created. This helps

prevent unintended side effects and makes code more robust.

- First-Class and Higher-Order Functions:
    - In Python, functions are first-class citizens, meaning they can be treated like any other object. They can be assigned to variables, passed as arguments to other functions, and returned from functions.
    - Higher-order functions are functions that [1] take other functions as arguments or return them as results.

1. medium.com

medium.com

- Recursion:

  Functional programming often uses recursion instead of loops for iteration.

- Lambda Functions:
    - These are small, anonymous functions

-
  - defined using the lambda keyword. They're useful for creating simple functions on the fly.

Python's Functional Programming Tools:

- map():
  - Applies a function to each item in an iterable (like a list) and returns an iterator of the results.
- filter():
  - Filters elements from an iterable based on a given function, returning an iterator of the filtered elements.
- reduce() (from functools):
  - Applies a rolling computation to sequential pairs of elements in an iterable, reducing them to a single value.
- List Comprehensions:
  - Provide a concise way to create lists by applying expressions to elements of existing iterables. They often serve as a functional alternative to loops.
- Generator Expressions:

    - Similar to list comprehensions, but they create generators, which produce values on demand, saving memory.
- functools Module:
    - Provides higher-order functions and tools for working with functions, such as reduce() and partial().
- itertools Module:
    - Provides tools for creating and working with iterators, which are essential for lazy evaluation in functional programming.

How it Works in Practice:

- Instead of using loops to modify data, functional Python code often uses map(), filter(), and list comprehensions to create new data structures based on transformations of existing ones.
- Lambda functions are used to create short, anonymous functions that can be passed as arguments to higher-order functions.

- Recursion is used to solve problems by breaking them down into smaller, self-similar subproblems.

Benefits of Functional Programming in Python:

- Improved Readability: Functional code can be more concise and easier to understand, especially for data transformations.
- Increased Testability: Pure functions are easier to test because they have no side effects.
- Enhanced Concurrency: Immutability and pure functions make it easier to write concurrent and parallel code.
- Code Reusability: Higher-order functions promote code reusability.

Q2. Using map(), reduce(), and filter() functions for processing data.

map(), reduce(), and filter() are powerful built-in functions in Python that facilitate functional programming paradigms. They are especially useful for processing data within iterables like lists, tuples, and sets. Here's how to use them effectively:

1. map() Function:

- Purpose: Applies a given function to each item in an iterable and returns an iterator that yields the results.
- Syntax: map(function, iterable, ...)

Example: Python

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

```
def to_upper(string):
return string.upper()
```

```
strings = ["hello", "world"]  upper_strings = map(to_upper, strings)
print(list(upper_strings)) #Output: ['HELLO', 'WORLD']
```

- Explanation:
  - The map() function applies the lambda function (or defined function) to each element of the numbers list.
  - It returns an iterator, so we convert it to a list to see the results.

2. filter() Function:

- Purpose: Filters elements from an iterable based on a given function (that returns True or False).
- Syntax: filter(function, iterable) □ Example:

Python

numbers = [1, 2, 3, 4, 5, 6] even_numbers = filter(lambda x: x % 2 == 0, numbers)

print(list(even_numbers)) # Output: [2, 4, 6]

words = ["apple", "banana", "orange", "kiwi"] long_words = filter(lambda word: len(word) > 5, words) print(list(long_words)) #output: ['banana', 'orange']

Explanation: ○ The filter() function applies the lambda function to each element of the numbers list.

○ It keeps only the elements for which the function returns True. ○ it also returns an iterator.

3. reduce() Function:

- Purpose: Applies a rolling computation to sequential pairs of elements in an iterable, reducing them to a single value.
- Syntax: reduce(function, iterable[, initializer])
- Note: reduce() is in the functools module, so you need to import it: from functools import reduce
- Example: Python from functools import reduce

numbers = [1, 2, 3, 4, 5] sum_of_numbers = reduce(lambda x, y: x + y, numbers)

print(sum_of_numbers) # Output: 15

multiplied_numbers = reduce(lambda x,y: x*y, numbers) print(multiplied_numbers) #output: 120

Explanation: ○ The reduce() function applies the lambda function to the first two elements of the numbers list. ○ Then, it applies the function to the result and the next element, and so on. ○ It reduces the list to a single value.

Key Advantages:

- Conciseness: These functions often allow you to write more compact and readable code compared to traditional loops.
- Functional Programming: They promote a functional programming style, which can lead to more maintainable and testable code.
- Efficiency: In many cases, these functions are implemented efficiently in C, making them faster than equivalent Python loops.

Important Considerations:

map() and filter() return iterators, so you'll usually need to convert them to lists or other data structures to view the results.

- reduce() is very powerful, but for simple sums or products, the built in sum() and prod() functions are often more readable.
- For very complex data processing pipelines, consider libraries like NumPy and Pandas, which offer more advanced and optimized tools.

Q3. Introduction to closures and decorators.

Closures and decorators are powerful features in Python that enhance code reusability, modularity, and flexibility. Here's an introduction to both:

1. Closures:

- Definition:
  - A closure is a function object that remembers values in enclosing scopes even if they are not present in memory.
  - In simpler terms, a closure "closes over" the environment in which it was created.
- How it Works:
  - Closures are created when a nested function references a variable from its enclosing scope.
  - The inner function retains access to the outer function's variables even after the outer function has finished executing.
- Example: Python

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

closure = outer_function(10)
result = closure(5)
# closure remembers x = 10
print(result)  # Output: 15
```

- Key Characteristics:
  - Nested function.
  - Inner function refers to a variable in the enclosing scope.
  - Outer function returns the inner function.

2. Decorators:

Definition: ◦ Decorators are a way to modify or extend the behavior of functions or classes without changing their source code. ◦ They "wrap" a function with another function, adding functionality before or after the original function is called.

- How it Works: ◦ Decorators use closures to achieve their functionality. ◦ They take a function as input, add some functionality, and return a modified function.
- Syntax: ◦ Decorators are applied using the @decorator_name syntax above a function definition.
- Example: Python def my_decorator(func):

def wrapper():

print("Something is happening before the function is called.")

func()

print("Something is happening after the function is called.") return wrapper

@my_decorator

def say_hello():

print("Hello!")

say_hello()

- Explanation: ○ my_decorator is a decorator function that takes a function func as input. ○ It defines a wrapper function that adds extra functionality before and after calling func. ○ @my_decorator applies the decorator to the say_hello function. ○ When say_hello is called, the wrapper function is executed.
- Decorators with Arguments:

- Decorators can also accept arguments, which adds another layer of flexibility. - Decorators that decorate functions with arguments, must also account for those arguments.

Key Benefits:

- Code Reusability: Decorators allow you to apply the same modification to multiple functions.
- Separation of Concerns: Decorators separate the core logic of a function from its auxiliary functionality.
- Improved Readability: Decorators can make code more concise and easier to understand.
- Metaprogramming: They enable you to write code that modifies or extends other code.

Closures and decorators are powerful tools for writing cleaner, more modular, and more flexible Python code.

LAB:

Q1. Write a Python program to apply the map() function to square a list of numbers.

def square_number(n): """Squares a given number.""" return n * n # Example usage: numbers = [1, 2, 3, 4, 5] # Apply the map() function to square each number in the list

squared_numbers = list(map(square_number, numbers)) # Print the result
print(squared_numbers) # Output: [1, 4, 9, 16, 25]
#Using a lambda function, which is more concise.
squared_numbers_lambda = list(map(lambda x: x * x, numbers)) print(squared_numbers_lambda) #Output: [1, 4, 9, 16, 25] #Another example with a different list. numbers2 = [10, 20, 30, 40]
squared_numbers2 = list(map(square_number, numbers2)) print(squared_numbers2) #output: [100, 400, 900, 1600]

Q2. Write a Python program that uses reduce() to find the product of a list of numbers.

from functools import reduce def multiply(x, y): """Multiplies two numbers."""" return x * y # Example usage: numbers = [1, 2, 3, 4, 5] # Use reduce() to find the product of the numbers product = reduce(multiply, numbers) # Print the result print(f"The product of the numbers is: {product}") # Output: 120 # Example using a lambda function (more concise) product_lambda = reduce(lambda x, y: x * y, numbers) print(f"The product (using lambda) is: {product_lambda}") #output: 120 #Example with a different list. numbers2 = [2,4,6] product2 = reduce(multiply, numbers2) print(f"The product of the second list is: {product2}") #output: 48

Q3. Write a Python program that filters out even numbers using the filter() function.

```
def is_odd(n): """Checks if a number is odd."""
return n % 2 != 0 # Example usage: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] # Use filter() to get only the odd numbers odd_numbers = list(filter(is_odd, numbers)) # Print the result print(f"Odd numbers: {odd_numbers}") # Output: [1, 3, 5, 7, 9] # Using a lambda function (more concise) even_numbers = list(filter(lambda x: x % 2 == 0, numbers)) print(f"Even numbers: {even_numbers}") #Output: [2, 4, 6, 8, 10] #Another example with a different list. numbers2 = [11, 22, 33, 44, 55] odd_numbers2 = list(filter(is_odd, numbers2)) print(f"Odd numbers from second list: {odd_numbers2}") #output: [11, 33, 55]
```