|| **श्रीराम** ||

Basic Python

| Q. **What is the difference between a list and a tuple?**  **Mutability**:   * **List**: Lists are mutable, meaning the elements within a list can be changed, added, or removed after the list has been created. * **Tuple**: Tuples are immutable, meaning once a tuple is created, it cannot be modified. You cannot add, remove, or change elements.   **Syntax**:   * **List**: Lists are defined using square brackets **[]**. Example: **my\_list = [1, 2, 3].** * **Tuple**: Tuples are defined using parentheses **()**. Example: **my\_tuple = (1, 2, 3).**   **Usage**:   * **List**: Lists are generally used when you need a collection of items that may need to be changed during the program's execution. For example, maintaining a dynamic list of student names. * **Tuple**: Tuples are used when you want a collection of items that should not change throughout the program. For example, representing a fixed geographical coordinate (latitude, longitude).   **Performance**:   * **List**: Lists have a slightly larger memory overhead because they are mutable and dynamic. * **Tuple**: Tuples are more memory efficient than lists because they are immutable.   **Methods**:   * **List**: Lists have several methods like **append(), extend(), remove()**, etc., that allow modification of the list. * **Tuple**: Tuples have only two methods: **count()** and **index()**, since their contents cannot be altered.   # List example  my\_list = [1, 2, 3]  print("Original list:", my\_list)  my\_list[0] = 10 # Modify the first element  print("Modified list:", my\_list)  # Output: [10, 2, 3]  my\_list.append(4) # Append a new element  print("After appending:", my\_list)  # Output: [10, 2, 3, 4]  # Tuple example  my\_tuple = (1, 2, 3)  print("Original tuple:", my\_tuple)  # Uncommenting the following lines will raise a TypeError because tuples are immutable  # my\_tuple[0] = 10  # my\_tuple.append(4)  # Tuples are ideal for storing values that should not change, like coordinates  coordinates = (40.7128, -74.0060)  print("Coordinates:", coordinates)  # Output: (40.7128, -74.0060)  **In Summary**: Use lists when you need a collection of elements that can change over time. Use tuples when you need a collection of elements that should remain constant throughout the program. |
| --- |

| Q. **How do you create a lambda function?**  A lambda function in Python is a small, anonymous function defined using the **lambda** keyword. Unlike regular functions defined with **def**, lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned when the function is called. Lambda functions are often used for short, simple operations and are typically used in contexts where a function is needed temporarily.  Syntax of a Lambda Function:  **lambda *arguments*: *expression***  arguments**:** The input(s) to the function (like parameters in a normal function).  expression**:** A single expression whose result will be returned by the lambda function. Example of a Lambda Function:  1. Basic Example:   # A lambda function to add 10 to a number  add\_10 = lambda x: x + 10  # Using the lambda function  result = add\_10(5)  print(result)  # Output: 15  2. Lambda Function with Multiple Arguments:  # A lambda function to multiply two numbers  multiply = lambda x, y: x \* y  # Using the lambda function  result = multiply(4, 5)  print(result)  # Output: 20  3. **Using Lambda Functions with Built-in Functions:**  Lambda functions are often used with functions like map(), filter(), and sorted().   * **Example with map()**: Applying a lambda function to each element in a list.   # Using lambda with map to square each number in a list  numbers = [1, 2, 3, 4]  squared\_numbers = list(map(lambda x: x \*\* 2, numbers))  print(squared\_numbers)  # Output: [1, 4, 9, 16]   * **Example with filter()**: Filtering elements in a list based on a condition.   # Using lambda with filter to select even numbers  numbers = [1, 2, 3, 4, 5, 6]  even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))  print(even\_numbers)  # Output: [2, 4, 6]  **Key Points to Remember**:   * Lambda functions are limited to a single expression and cannot contain statements or multiple expressions. * They are best suited for short, simple operations and can be used wherever a function object is required. * Although they provide a concise way to create small functions, using them for complex operations can make code less readable. |
| --- |

| Q. What is the difference between deepcopy and shallowcopy?  In Python, **deepcopy** and **shallowcopy** are two functions used to create copies of objects. They differ in how they handle nested data structures.  **shallowcopy:**   * Creates a new object and copies the references to the original object's elements. * If the original object contains mutable data structures (like lists or dictionaries), changes made to the original object will also reflect in the shallow copy.   **deepcopy:**   * Creates a new object and recursively copies all elements, including nested data structures. * Changes made to the original object will not affect the deep copy.   Example:  import copy  # Original list  original\_list = [[1, 2], 3, 4]  # Shallow copy  shallow\_copy = copy.copy(original\_list)  # Deep copy  deep\_copy = copy.deepcopy(original\_list)  # Modify the original list  original\_list[0][0] = 10  # Print the results  print("Original list:", original\_list)  print("Shallow copy:", shallow\_copy)  print("Deep copy:", deep\_copy)  Output:  Original list: [[10, 2], 3, 4]  Shallow copy: [[10, 2], 3, 4]  Deep copy: [[1, 2], 3, 4]  As you can see, the shallow copy is affected by the changes made to the original list because it shares the same nested list. On the other hand, the deep copy remains unchanged because it has its own independent copy of the nested list.  **When to use which:**   * **shallowcopy:** Use when you only need a copy of the top-level object and don't care about changes to nested data structures. * **deepcopy:** Use when you need a complete independent copy of the object, including all nested data structures. |
| --- |

| Q. Describe list comprehensions and give an example.  **List Comprehensions** in Python provide a concise way to create lists. It allows you to generate a new list by applying an expression to each item in an existing iterable (like a list, tuple, or range) and, optionally, include a condition for filtering items.  Syntax of List Comprehension  [expression **for** item **in** iterable **if** condition]   * expression: The operation or transformation applied to each item. * item: The variable representing each element in the iterable. * iterable: The collection you are iterating over (such as a list or range). * condition (optional): An optional filter that determines whether to include the item in the new list.  **Example of List Comprehension** Let’s say we have a list of numbers, and we want to create a new list with the squares of those numbers.  **Traditional Approach using a for loop:**  numbers = [1, 2, 3, 4, 5]  squares = []  for number in numbers:  squares.append(number \*\* 2)  print(squares)  # Output: [1, 4, 9, 16, 25]  **Using List Comprehension:**  numbers = [1, 2, 3, 4, 5]  squares = [number \*\* 2 for number in numbers]  print(squares)  # Output: [1, 4, 9, 16, 25] **Explanation**  * **Expression**: number \*\* 2 is the expression that calculates the square of each number. * **for loop**: for number in numbers iterates through each item in the numbers list. * **Condition**: (None in this example).  **Example with a Condition** Let's create a list of squares for only even numbers in the list.  **Using List Comprehension with a Condition:**  numbers = [1, 2, 3, 4, 5]  even\_squares = [number \*\* 2 for number in numbers if number % 2 == 0]  print(even\_squares)  # Output: [4, 16]  **Condition**: if number % 2 == 0 filters the list to include only even numbers. **Benefits of List Comprehension**  * **Conciseness**: More compact and readable code compared to using traditional loops. * **Performance**: Generally faster than equivalent for loops due to optimization at the Python interpreter level. * **Expressiveness**: Makes it easier to express operations on lists, making the code easier to understand and maintain.   List comprehensions are a powerful feature in Python for creating lists in a clean and concise manner. |
| --- |

| Q. What does the enumerate() function do?  The **enumerate()** function in Python adds a counter to an iterable (like a list, tuple, or string) and returns it as an enumerate object. This allows you to loop over the iterable and have both the index (or position) and the value of each item available during each iteration. This is particularly useful when you need to keep track of the index while iterating through a collection. **Syntax of enumerate()** enumerate(iterable, start=0)   * **iterable**: The collection you want to loop over (e.g., list, tuple, string). * **start** (optional): The starting index of the counter. By default, it is set to 0.  **Example of enumerate() in Use** Suppose you have a list of fruits, and you want to print each fruit along with its index:  fruits = ['apple', 'banana', 'cherry']  for index, fruit in enumerate(fruits):  print(index, fruit)  Output:  0 apple  1 banana  2 cherry  In this example, enumerate(fruits) returns an enumerate object that produces pairs containing each index and its corresponding fruit. **Example with a Starting Index** You can also specify a different starting index by using the start parameter:  python  Copy code  fruits = ['apple', 'banana', 'cherry']  for index, fruit in enumerate(fruits, start=1):  print(index, fruit)  **Output:**  1 apple  2 banana  3 cherry  Here, the start=1 argument makes the index start from 1 instead of the default 0. **Benefits of Using enumerate()**  1. **Readability**: Improves code readability by providing a clean and concise way to access both the index and the value in loops. 2. **Efficiency**: More efficient and Pythonic than using a manual counter with a for loop. 3. **Convenience**: Automatically handles the indexing for you, reducing the likelihood of off-by-one errors.   Using enumerate() is especially helpful when you need to keep track of an index and the value in loops without having to manually update a counter. |
| --- |

| Q. How do you handle exceptions in Python?  In Python, exceptions are handled using the **try, except, else,** and **finally** blocks. This structure allows you to write code that responds to errors or exceptions that may occur during program execution, preventing the program from crashing and providing a way to handle errors gracefully. **Basic Structure of Exception Handling** try:  # Code that might raise an exception  risky\_code()  except SomeException as e:  # Code that runs if the specified exception occurs  handle\_error(e)  else:  # Code that runs if no exception occurred in the try block  no\_error\_occurred()  finally:  # Code that always runs, regardless of whether an exception occurred  cleanup\_code() **Key Components:**  1. **try Block**:    * Contains code that might raise an exception. If an exception occurs, the code in the try block stops executing and moves to the corresponding except block. 2. **except Block**:    * Catches and handles the exception that occurs in the try block. You can specify the type of exception you want to catch. If an exception is raised, and there is an except block for it, the code inside this block runs.    * You can also handle multiple exceptions by specifying different except blocks. 3. **else Block** (Optional):    * Runs only if no exceptions were raised in the try block. This is useful for code that should execute only when everything goes smoothly. 4. **finally Block** (Optional):    * Executes regardless of whether an exception occurred or not. This block is often used for cleanup actions like closing files or releasing resources.  **Example of Exception Handling** Here is an example that demonstrates how to use each block:  try:  # Code that might raise an exception  num = int(input("Enter a number: "))  result = 10 / num  except ValueError as ve:  # Handle a specific exception (ValueError)  print("You must enter a valid number.", ve)  except ZeroDivisionError as zde:  # Handle another specific exception (ZeroDivisionError)  print("Cannot divide by zero.", zde)  else:  # Runs only if no exceptions occur in the try block  print("The result is:", result)  finally:  # Always runs, regardless of exceptions  print("Execution completed.") **Common Exception Handling Tips**  1. **Catch Specific Exceptions**: Instead of catching all exceptions with a general except statement, it’s best practice to catch specific exceptions to make the code easier to debug and maintain. 2. **Use finally for Cleanup**: The finally block is ideal for releasing resources or performing cleanup actions, such as closing files or network connections, which should occur regardless of whether an error occurred. 3. **Avoid Swallowing Exceptions**: Be cautious when catching exceptions. Catching all exceptions with a bare except clause can hide errors and make debugging difficult. Always aim to handle exceptions explicitly and meaningfully. 4. **Re-raising Exceptions**: Sometimes, after handling an exception, you might want to re-raise it to propagate it further up the call stack. This can be done using the raise keyword.   try:  # some code that might throw an exception  risky\_operation()  except SomeException as e:  # handle exception  print("Handling exception:", e)  # re-raise exception  raise  By properly handling exceptions, you can write robust Python programs that gracefully handle unexpected situations. |
| --- |

| Q. What is the difference between global and local variables? **Global Variables** **Definition**: Global variables are declared outside of any function, typically at the top level of a script. They are accessible from any part of the code, including within functions.  **Scope**: They have a global scope, meaning they can be accessed and modified from anywhere in the program, as long as they are not shadowed by local variables with the same name.  **Lifetime**: They exist for the entire duration of the program's execution. As long as the program is running, global variables retain their value.  **Usage**: Use global variables for values that need to be shared across multiple functions or throughout the entire program.  **Modification**: To modify a global variable inside a function, you must declare it as global using the global keyword.  **Example**:  x = 10 # This is a global variable  def print\_global():  print(x) # Accesses the global variable x  print\_global() # Output: 10  print(x)  # Output: 10  In this example, **x** is a global variable because it is defined outside any function. It is accessible both inside and outside the **print\_global** function. **Local Variables** **Definition**: Local variables are declared within a function or block of code. They are only accessible within that function or block.  **Scope**: They have a local scope, meaning they can only be accessed within the function or block where they are defined. They are not visible outside of that function or block.  **Lifetime**: They exist only while the function or block is executing. Once the function or block finishes execution, the local variable is destroyed and its value is lost.  **Usage**: Use local variables for temporary values or computations that are specific to a particular function.  **Modification**: Local variables can be modified directly within the function where they are defined.  **Example**:  def print\_local():  y = 5 # This is a local variable  print(y)  print\_local() # Output: 5  print(y) # Raises a NameError because y is not accessible outside the function  In this example, **y** is a local variable because it is defined inside the **print\_local** function. It is only accessible within that function. Trying to access y outside the function results in an error. **Modifying Global Variables Inside Functions** If you need to modify a global variable inside a function, you must declare it as global within that function using the **global** keyword:  **Example**:  z = 20 # This is a global variable  def modify\_global():  global z # Declares that we are using the global variable z  z = 30  modify\_global()  print(z)  # Output: 30  In this example, the **global** keyword allows the function **modify\_global** to modify the global variable **z**. Without the **global** keyword, the assignment **z = 30** would create a new local variable **z** instead of modifying the global one.  Understanding these differences helps in managing variable scope and lifetime effectively, ensuring that variables are used and modified appropriately throughout your Python programs. |
| --- |

| Q. How can you convert a string representation of a number into an integer?  To convert a string representation of a number into an integer in Python, you use the built-in int() function. This function takes a string (or another number) and converts it into an integer. Here’s how you can do it: **Basic Conversion** If you have a string that represents a number, you can convert it to an integer like this:  python  Copy code  number\_str = "123"  number\_int = int(number\_str)  print(number\_int) # Output: 123  print(type(number\_int)) # Output: <class 'int'> **Handling Different Bases** The int() function can also handle strings representing numbers in different bases. By default, int() assumes the string is in base 10 (decimal). However, you can specify a different base if needed:  python  Copy code  # Convert a binary (base 2) string to an integer  binary\_str = "1010"  binary\_int = int(binary\_str, 2)  print(binary\_int) # Output: 10  # Convert a hexadecimal (base 16) string to an integer  hex\_str = "1a"  hex\_int = int(hex\_str, 16)  print(hex\_int) # Output: 26  In the above examples:   * For the binary string "1010", specifying base 2 converts it to decimal 10. * For the hexadecimal string "1a", specifying base 16 converts it to decimal 26.  **Handling Errors** If the string does not represent a valid number or is in an invalid format, the int() function will raise a ValueError. You can handle this exception using a try and except block:  python  Copy code  number\_str = "abc" # Invalid number string  try:  number\_int = int(number\_str)  except ValueError:  print("The string does not represent a valid number.")  In this example, "abc" is not a valid number string, so the ValueError is caught, and a message is printed. **Summary** To convert a string to an integer:   * Use int(string) for decimal numbers. * Use int(string, base) to convert strings in other bases (binary, hexadecimal, etc.). * Handle possible errors with a try and except block if the string might not be a valid number.   This method is straightforward and effective for converting numerical string representations into integer types in Python. |
| --- |

1. Describe the use of the \*args and \*\*kwargs parameters.
2. What is the purpose of the \_\_init\_\_ method in a class?

String Manipulation

Algorithms