1. What is the relationship between def statements and lambda expressions ?

**Ans:-** The relationship between ‘def’ statement and lambda expression in python is that they both define functions. However, there are a few key differences between them.

1. **For Def statement :- def function\_name(parameters):**

**# Function body**

**for Lambda expression :- lambda parameters: expression**

1. Function name :- When using a ‘def’ statement , you must provide a name for the function name, whereas the lambda expressions are anonymous functions and do not have a name.
2. Function body :- ‘def’ statement allow you to write multiple statements in the function body, whereas lambda expression can only contain a single expression.
3. Return value :- Functions defined using ‘def’ statement use the ‘return’ statement to specify the return value explicitly. In contrast, lambda expressions implicitly return the value of the function they contain.
4. Scope :- Functions defined with ‘def’ statements have their own local namespace, which means they can access variables from the surrounding scope and modify them if necessary. Lambda expressions are limited to their own expression scope and do not have access to variables outside their scope.

2. What is the benefit of lambda?

**Ans:-** The lambda function in python provides several benefits including :

1. Consciousness :- Lambda function allow you to define small , anonymous functions in a single line of code. They eliminate the need of writing full ‘def’ statement, making the code more compact and easy to read.
2. Readability :- Lambda expression can make the code more readable in certain situations. When used appropriately, they can provide clear and concise representation of a simple function, reducing the cognitive load on the reader.
3. Convenience :- Lambda expressions are convenient for creating and using functions on the fly. They can be used directly as arguments to higher-order functions such as ‘map()’, ‘filter()’ or ‘sort()’ eliminating the need to define separate functions for a one time operation.
4. Functional programming:- Lambda expressions are commonly used in functional programming paradigms. They allow you to define functions as first-class objects, which can be passed around, , stored in data structures, and used in higher order functions. This promotes a more functional and expressive coding style.
5. Focus on the essence :- Lambda expression help in focusing on the essence of a functions or operation without getting distracted by the details of the function naming and definition. They allow you to express the logic or transformation directly without the need of additional lines of code.

3. Compare and contrast map, filter, and reduce.

**Ans:-** The functions ‘map()’, ‘filter()’ and ‘reduce()’ are the higher order functions in python, that operate on sequences such as lists, tuples or strings and perform specific operations on the elements of the sequence. While they have some similarities, they differ in their purpose and functionality:-

1. **Map(function, iterable) :-** The map function applies a given function to each item in an iterable and return an iterator that yields the result. It transforms each item of the iterable by applying the specified function and returns an iterator with the transformed value in the same order.
2. **Filter(function, iterable) :-** The ‘filter()’ function applies a given function to each item in an iterable and returns an iterator that yields the item for which the function returns ‘true’. It filter the elements of the iterable based on a specific condition defined by the function and returns an iterator with the filtered elements.
3. **Reduce(function, iterable[, initializer]) :-** The reduce function applies a given function to the elements of an iterable in a cumulative way, reducing them to a single value. It repeatedly applies the specified function to pairs of elements from the iterable, accumulating the result. The final result is a single value.

4. What are function annotations, and how are they used?

**Ans:-** Function annotations in python are a way to associate metadata or additional information with the parameters and return value of a function. They provide a way to specify the expected types of the function arguments and return type. Function annotations are optional and do not affect the runtime behavior of the function. Function annotations are defined using colons(‘:’) after the parameter names and return arrows(‘’->”) before the return type. The annotations can be any valid python expression, but commonly used types or classes are used to specify the types. Here’s an example of a function with annotations:-

**def add(x: int, y: int) -> int:**

**return x + y**

Annotations can be also be used with default parameter values and variable length arguments:-

**def multiply(x: int, y: int = 1, \*args: float) -> float:**

**result = x \* y**

**for arg in args:**

**result \*= arg**

**return result**

5. What are recursive functions, and how are they used?

**Ans:-** Recursive functions are functions in programming that call themselves within their own definition. They are used to solve problems by breaking them down into smaller, simpler instances of the same problem until a base case is reached. The base case is a condition that determines when the recursion should stop and the function should return a specific result. Here’s an example of a recursive function that calculates the factorial of a number:-

**def factorial(n):**

**if n == 0:**

**return 1**

**else:**

**return n \* factorial(n - 1)**

6. What are some general design guidelines for coding functions?

**Ans:-** When coding functions, , following general design guidelines can help improve code readability, maintainability, and reusability. Here are some key guidelines to consider:-

1. Function purpose and single responsibility :- Functions should have a clear and well defined purpose, focusing on performing a single task or responsibility. This helps improve code organization and make functions more modular and reusable.
2. Functional naming :- Choose descriptive and meaningful names for functions that accurately reflect their purpose or action. Use verbs or verb phrases to indicate what the function does. Aim for clarity and avoid ambiguous or overly generic names.
3. Function length and complexity :- Keep function concise and focused. Avoid excessively long functions that perform multiple unrelated tasks. Split complex functions into smaller, more manageable functions to improve readability and maintainability.
4. Function parameters :- Minimize the number of parameters passed to a function. Aim for a small and meaningful set of parameters that are necessary for the functions operation. Excessive parameters can make functions harder to understand and increase coupling with the calling code.
5. Function return values :- Function should have a well defined return value or purpose. Use meaningful return values or consider returning None If a function doesn’t have a specific results to return. Avoid relying on modifying mutable objects as a way to communicate results.

7. Name three or more ways that functions can communicate results to a caller.

**Ans:-** Functions in python can communicate results to a caller through various mechanisms. Here are three common ways:-

1. **Return statement :-** The most straightforward way for a function to communicate a result is through the ‘return’ statement. The function calculates or processes the desired results and uses the ‘return’ statement to send the result back to the caller. The caller can capture and use the returned value as needed.

**def add(a, b):**

**return a + b**

**result = add(3, 5)**

**print(result) # Output: 8**

1. **Modifying mutable objects :-** Functions can communicate results by modifying mutable objects that are passed as arguments. In this case, the function doesn’t explicitly return a value but instead modifies the state of the object, which can be observed by the caller.

**def square\_list(numbers):**

**for i in range(len(numbers)):**

**numbers[i] = numbers[i] \*\* 2**

**my\_list = [1, 2, 3, 4]**

**square\_list(my\_list)**

**print(my\_list) # Output: [1, 4, 9, 16]**

1. **Global variables:-** Although generally not recommended, functions can communicate results through global variables. Global variables are accessible to all parts of the codebase, including functions. By modifying or assigning a value to a global variable within a function, the result can be accessed by the caller.

**result = None**

**def calculate():**

**global result**

**result = 42**

**calculate()**

**print(result) # Output: 42**