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

Ans : “def” used to create named functions that can be reused throughout the code. Can contain multiple expressions and statements, including loops, conditionals, and other complex logic.

def add(a, b):

return a + b

“lambda” used to create anonymous functions, typically for short, simple operations. Often used in functional programming contexts like passing functions as arguments to higher-order functions (e.g map, filter or sorted). Limited to a single expression. Cannot contain statements or multiple expressions.

add = lambda a, b: a + b

1. What is the benefit of lambda?

Ans:

Conciseness: Lambda expressions allow for the creation of small, single-use functions without the need for formally defining a function using the def statement. This can make the code more concise and readable in certain situations.

Inline Use : Lambdas are often used where functions are required for a short period of time, such as in argument lists for higher-order functions like map, filter, and sorted. This can be more convenient than defining a function separately.

Anonymous Functions : lambda functions are anonymous (i.e., they don’t have a name), they are useful for creating quick, throwaway functions that don’t need to be referenced by name elsewhere in the code.

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

Ans . **map**  is best for transforming each element in an iterable.

numbers = [1, 2, 3, 4, 5]

squared\_numbers = list(map(lambda x: x \* x, numbers))

# Output: [1, 4, 9, 16, 25]

**Filter** Selects elements from an iterable that meet a condition.

numbers = [1, 2, 3, 4, 5]

even\_numbers = list(filter(lambda x : x%2==0 , numbers))

# Output: [2, 4]

**reduce:** Aggregates all elements in an iterable into a single result.

from functools import reduce

numbers = [1, 2, 3, 4, 5]

summation = reduce(lambda x, y: x + y, numbers)

# Output: 15

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

Ans : Function annotations are optional additions to function definitions in Python that provide extra information about the function's arguments and return type. Function annotations, while not mandatory, are a valuable tool for enhancing code readability, promoting static type checking, and improving the overall developer experience in Python.

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

Ans : Recursive functions are functions that call themselves in order to solve smaller instances of the same problem.

**Common uses of recursive functions:**

Tree traversal: Recursively traverse through the nodes of a tree data structure (e.g., depth-first search, breadth-first search).

Factorial calculation: Calculate the factorial of a number by multiplying it by the factorial of the previous number (until reaching the base case of 1).

Fibonacci sequence: Generate the Fibonacci sequence by recursively defining a number as the sum of the two preceding numbers.

Problem-solving with divide-and-conquer: Divide a complex problem into smaller, solvable subproblems using recursion, then combine the solutions of the subproblems to get the solution of the original problem (e.g., merge sort, quicksort).

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

Ans.

* Readability and Maintainability: Focus on writing clear and understandable code. Use meaningful names, follow consistent formatting, and add comments for complex logic.
* Modularity and Reusability: Break down complex problems into smaller, well-defined functions that can be reused throughout your codebase.
* Efficiency and Performance: Consider the time and space complexity of your functions, especially when dealing with large datasets. Avoid unnecessary operations.
* Error Handling: Validate function inputs and handle errors gracefully by providing informative messages.
* Testing: Write unit tests to ensure your functions work correctly under different input conditions.

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

Ans : The most common way for a function to communicate results to the caller is by using the return statement. This allows the function to send a value (or multiple values) back to the caller.

def add(a, b):

return a + b

result = add(5, 3)

print(result)

# Output: 8

A function can modify mutable objects (such as lists or dictionaries) passed as arguments, effectively using them as output parameters. This allows the function to "return" multiple results or large amounts of data.

def update\_list(my\_list):

my\_list.append(1)

my\_list.append(2)

my\_list.append(3)

numbers = []

update\_list(numbers)

print(numbers)

# Output: [1, 2, 3]

Functions can communicate results by raising exceptions to signal that an error or a special condition has occurred. The caller can handle these exceptions using try and except blocks.

def divide(a, b):

if b == 0:

raise ValueError("Cannot divide by zero")

return a / b

try:

result = divide(10, 0)

except ValueError as e:

print(e)

# Output: Cannot divide by zero