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

Both def statements and lambda expressions in Python are used to define functions. The primary difference between them lies in their syntax and typical use cases:

def statements: Used to create named functions, which can include multiple statements and are generally more complex. Functions defined with def can have a docstring, and are stored with a name in the local namespace.

lambda expressions: Used to create anonymous functions, which are typically simple, single-expression functions. These are used in places where small throwaway functions are needed and are written using a more compact syntax.

2. What is the benefit of lambda?

The primary benefit of lambda expressions is that they allow for the creation of small, anonymous functions in a compact and readable way, especially when passing functions as arguments to higher-order functions (like map, filter, and reduce). They can make the code more concise and are useful in contexts where the function is used only once or a limited number of times.

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

map: Applies a given function to each item of an iterable (like a list) and returns a map object (an iterator).

nums = [1, 2, 3, 4]

squares = map(lambda x: x \*\* 2, nums) # Output: [1, 4, 9, 16]

filter: Filters items out of an iterable based on a function that returns either True or False for each item. Returns an iterator.

nums = [1, 2, 3, 4]

even\_nums = filter(lambda x: x % 2 == 0, nums) # Output: [2, 4]

reduce: Applies a binary function to the items of an iterable, cumulatively, to reduce it to a single value. The function should take two arguments.

from functools import reduce

nums = [1, 2, 3, 4]

sum = reduce(lambda x, y: x + y, nums) # Output: 10

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

Function annotations are a way of attaching metadata to function arguments and return values. They are purely for informational purposes and do not affect the function's behavior.

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

return x + y

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

Recursive functions are functions that call themselves within their definition to solve a problem by breaking it down into smaller, more manageable sub-problems.

def factorial(n):

if n == 1:

return 1

else:

return n \* factorial(n - 1)

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

Single Responsibility Principle: Each function should have a single, well-defined purpose.

Descriptive Naming: Use clear and descriptive names for functions and parameters.

Avoid Side Effects: Functions should avoid modifying global state or mutable arguments unless explicitly intended.

Keep Functions Short: Smaller functions are easier to understand, test, and maintain.

Use Docstrings: Document the purpose, arguments, return value, and any side effects of the function.

Parameter Defaults: Provide default values for parameters when appropriate to make the function more flexible and easier to use.

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

Return Values: The most common way, using the return statement.

def add(a, b):

return a + b

Output Parameters: Modify the arguments passed to the function (less common in Python).

def add\_to\_list(lst, item):

lst.append(item)

Exceptions: Raise exceptions to signal errors or exceptional conditions.

def divide(a, b):

if b == 0:

raise ValueError("Cannot divide by zero")

return a / b

Printing: Output information directly to the console (though typically avoided in production code).

def greet(name):

print(f"Hello, {name}!")