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

ANSWER.

- Both `def` statements and `lambda` expressions are used to define functions, but they serve different purposes.

- `def` statements are used for creating named functions with multiple statements, while `lambda` expressions are used for creating anonymous functions with a single expression.

- `def` statements provide more flexibility and readability, especially for complex functions, while `lambda` expressions are more concise and are often used for short, simple operations.

2. What is the benefit of lambda?

ANSWER.

Lambda expressions provide a convenient and concise way to define small, anonymous functions inline, which can lead to more readable and expressive code, especially in functional programming scenarios and situations where small, one-time functions are needed.

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

ANSWER.

- `map()` applies a function to each element of an iterable and returns an iterator of the results, while `filter()` selects elements from an iterable based on a given condition and returns an iterator of the selected elements.

- Both `map()` and `filter()` produce iterators as output, so they can be consumed lazily.

- `reduce()` applies a function of two arguments cumulatively to the items of an iterable to reduce it to a single value. It requires a binary function and an iterable as input, and it reduces the iterable to a single value by applying the function sequentially.

- `reduce()` is not built into Python's core library and must be imported from the `functools` module in Python 3.

- While `map()` and `filter()` are commonly used for element-wise transformations and filtering, `reduce()` is typically used for operations like summing, product computation, finding the maximum or minimum value, etc., where the result depends on all elements of the iterable.

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

ANSWER.

Function annotations are flexible and can be used for various purposes beyond type hints, such as documenting the behavior of parameters or indicating units of measurement. However, their interpretation and usage depend on conventions and external tools, as Python itself does not enforce or enforce any particular meaning or behavior for annotations.

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

ANSWER.

Recursive functions are functions that call themselves directly or indirectly in order to solve a problem. They are used to break down a complex problem into smaller, simpler subproblems, which are then solved recursively.

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

ANSWER.

Here are some general design guidelines for coding functions:

1. Single Responsibility Principle (SRP): Each function should have a single responsibility or do one thing well. This makes functions easier to understand, test, and maintain. If a function is doing too much, consider breaking it down into smaller, more focused functions.

2. Descriptive and Meaningful Names: Choose descriptive and meaningful names for functions that accurately convey their purpose and behavior. A well-named function should provide a clear indication of what it does without needing to look at its implementation.

3. Consistent Naming Conventions: Use consistent naming conventions for functions, parameters, and variables throughout your codebase. This makes the code more readable and reduces cognitive overhead for developers.

4. Proper Documentation: Provide clear and concise documentation for each function, including its purpose, parameters, return values, and any side effects. Docstrings are a common way to document functions in Python, but other documentation formats may be used depending on the programming language and project requirements.

5. Use Parameters Instead of Global Variables: Avoid using global variables within functions whenever possible, as they can lead to unintended side effects and make functions harder to understand and test. Instead, pass parameters to functions to provide input and return values to provide output.

6. Keep Functions Short and Simple: Aim to keep functions short and focused, ideally no more than a few dozen lines of code. Long functions are harder to understand, test, and maintain, and they often indicate that the function is doing too much.

7. Avoid Side Effects: Minimize or eliminate side effects within functions, such as modifying global state, modifying mutable objects passed as arguments, or performing I/O operations. Functions with side effects are harder to reason about and can lead to unexpected behavior.

8. Error Handling and Validation: Include proper error handling and input validation within functions to handle unexpected conditions gracefully. This helps prevent runtime errors and improves the robustness and reliability of your code.

9. Avoid Magic Numbers and Constants: Avoid hardcoding magic numbers and constants within functions. Instead, define them as named constants or parameters with descriptive names to improve readability and maintainability.

10. Use Functional Decomposition: Break down complex tasks into smaller, more manageable functions that can be composed together to achieve the desired result. This promotes code reuse and modularity and makes the codebase easier to understand and maintain.