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

* **def statements**:
  + Used to define named functions.
  + Can contain multiple statements and complex logic.
  + Support annotations, docstrings, and return values explicitly.
  + Example:

def add(a, b):

return a + b

* **lambda expressions**:
  + Used to define small, anonymous functions.
  + Limited to a single expression (no statements or complex logic).
  + Do not support annotations or docstrings.
  + Example:

add = lambda a, b: a + b

**Relationship**:

* Both def and lambda are used to define functions.
* lambda is a shorthand for simple, one-line functions, while def is used for more complex functions.

**2. What is the benefit of lambda?**

* **Conciseness**: lambda allows you to define small functions in a single line.
* **Anonymous**: Functions can be used without assigning them a name.
* **Inline usage**: Useful for short, throwaway functions, especially in higher-order functions like map, filter, and reduce.

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

* **map**:
  + Applies a function to all items in an iterable.
  + Returns an iterator of results.
  + Example:

result = map(lambda x: x \* 2, [1, 2, 3])

print(list(result)) # Output: [2, 4, 6]

* **filter**:
  + Filters items in an iterable based on a condition.
  + Returns an iterator of items that satisfy the condition.
  + Example:

result = filter(lambda x: x > 2, [1, 2, 3, 4])

print(list(result)) # Output: [3, 4]

* **reduce**:
  + Applies a function cumulatively to items in an iterable, reducing it to a single value.
  + Requires importing from functools.
  + Example:

from functools import reduce

result = reduce(lambda x, y: x + y, [1, 2, 3, 4])

print(result) # Output: 10

**Comparison**:

* map transforms all items in an iterable.
* filter selects items based on a condition.
* reduce combines all items into a single value.

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

* **Function annotations**:
  + Provide metadata about the types of arguments and return values.
  + Defined using : for arguments and -> for return values.
  + Example:

def add(a: int, b: int) -> int:

return a + b

* **Usage**:
  + Annotations are stored in the \_\_annotations\_\_ attribute of the function.
  + They are not enforced by Python but can be used by tools like linters, type checkers, or IDEs.

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

* **Recursive functions**:
  + Functions that call themselves to solve smaller instances of the same problem.
  + Must have a base case to terminate the recursion.
  + Example:

def factorial(n):

if n == 0:

return 1

return n \* factorial(n - 1)

* **Usage**:
  + Useful for problems that can be broken down into smaller, similar subproblems (e.g., tree traversal, factorial, Fibonacci).

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

* **Single Responsibility Principle**: A function should perform one task.
* **Descriptive Names**: Use meaningful names for functions and variables.
* **Avoid Side Effects**: Functions should not modify global state or input arguments unless explicitly intended.
* **Keep Functions Short**: Aim for small, reusable functions.
* **Use Docstrings**: Document the purpose, parameters, and return values of functions.
* **Limit Parameters**: Avoid too many parameters; use data structures if necessary.
* **Handle Errors**: Use exceptions for error handling.

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

1. **Return Statement**:
   * Directly return a value or object.
   * Example:

def add(a, b):

return a + b

1. **Modifying Mutable Arguments**:
   * Modify mutable objects (e.g., lists, dictionaries) passed as arguments.
   * Example:

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

lst.append(item)

1. **Global Variables**:
   * Modify global variables (not recommended unless necessary).
   * Example:

result = 0

def set\_result(value):

global result

result = value

1. **Yield Statement (Generators)**:
   * Use yield to return a sequence of values over time.
   * Example:

def generate\_numbers():

for i in range(3):

yield i

1. **Exceptions**:
   * Raise exceptions to signal errors or special conditions.
   * Example:

def divide(a, b):

if b == 0:

raise ValueError("Cannot divide by zero")

return a / b