PYTHON ASSIGNMENT - 24

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

2. What is the benefit of lambda?

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

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

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

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

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

ANSWERS

1. Relationship between def statements and lambda expressions:

- Both **def** statements and **lambda** expressions are used to define functions in Python.
- **def** is a statement used for creating named functions, and it allows for more complex function definitions with multiple statements and a formal function body.
- **lambda** is an expression used for creating anonymous functions (functions without a name). It is often used for short, simple operations and allows for the creation of functions in a single line.

2. **Benefits of lambda:**

- Conciseness: Lambda expressions are concise and can be written in a single line.
- Readability: Lambda is useful for creating small, one-time-use functions without the need for a full function definition.

- Functional programming: Lambda is often used in functional programming constructs like **map**, **filter**, and **reduce**.

3. **Comparison of map, filter, and reduce:**
   - **map(function, iterable):** Applies the specified function to all items in the iterable and returns an iterator of the results.
   - result = map(lambda x: x**2, [1, 2, 3, 4]) # [1, 4, 9, 16]
   - **filter(function, iterable):** Filters items in the iterable based on the specified function and returns an iterator of the items that evaluate to **True**.
   - result = filter(lambda x: x % 2 == 0, [1, 2, 3, 4]) # [2, 4]
   - **reduce(function, iterable[, initializer]):** Applies the specified function cumulatively to the items in the iterable, reducing it to a single accumulated result.
   - from functools import reduce
   - result = reduce(lambda x, y: x + y, [1, 2, 3, 4]) # 10

4.Function annotations:

- Function annotations are a way to attach metadata to the parameters and return value of a function.
- They are defined using a colon (**:**) followed by an expression after the parameter or return type.
- Annotations are optional and don't affect the function's behavior; they provide additional information for documentation or type checking.
- def add(x: int, y: int) -> int:
- return x + y

5.Recursive functions:

- Recursive functions are functions that call themselves during their execution.
- They are often used for solving problems that can be broken down into smaller, similar subproblems.
- A base case is essential to prevent infinite recursion.
- def factorial(n):
- if n == 0:
- return 1
- else:
- return n * factorial(n-1)

6. Design guidelines for coding functions:

- Follow a clear naming convention that describes the function's purpose.
- Keep functions modular and focused on a single task (single responsibility principle).
- Use function parameters for input and return statements for output.
- Document functions with docstrings to explain their purpose, parameters, and return values.
- Consider function immutability and avoid side effects whenever possible.

7. Ways functions can communicate results to a caller:

- **Return statements:** Functions can use **return** statements to send a value back to the caller.
- **Global variables:** Functions can modify or use global variables to communicate information.
- **Mutable objects:** Functions can modify mutable objects (e.g., lists) that are passed as arguments.
- **Function parameters:** Functions can communicate information through parameters passed by the caller.
- **Exception handling:** Functions can raise exceptions to indicate errors or special conditions.