Assignment-24

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

Ans. Relationship between def statements and lambda expressions: Both def statements and lambda expressions create function objects, but with key differences:

```
# def statement - creates a named function
def add_numbers(x, y):
    """Add two numbers and return the result."""
    return x + y

# lambda expression - creates an anonymous function
add_lambda = lambda x, y: x + y

# Both can be used similarly
print(add_numbers(5, 3))  # Output: 8
print(add_lambda(5, 3))   # Output: 8
```

Key differences:

i. Lambda is limited to a single expression

ii. Lambda creates an anonymous function

iii. def can have docstrings and annotations, lambda cannot

iv. def allows for more complex function bodies

2. What is the benefit of lambda?

Ans.

- Compact one-liner functions
- Useful for short operations passed to higher-order functions
- Creates anonymous functions "on the fly"
- Ideal for simple transformations in functional programming constructs

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

Ans.

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
# map: applies a function to every item in an iterable
squared = list(map(lambda x: x**2, numbers))
print(f"map result: {squared}")  # Output: [1, 4, 9, 16, 25]

# filter: creates a list of elements for which a function returns True
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(f"filter result: {evens}")  # Output: [2, 4]

# reduce: applies a function of two arguments cumulatively to the items of a sequence
sum_all = reduce(lambda x, y: x + y, numbers)
print(f"reduce result: {sum_all}")  # Output: 15 (1+2+3+4+5)
```

Key differences:

i. map transforms elements

ii. filter selects elements

iii. reduce aggregates elements into a single value

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

Ans. Function annotations: Function annotations are optional metadata about the types used by user-defined functions.

```
def calculate_bmi(weight: float, height: float) -> float:
"""
Calculate BMI using weight in kg and height in meters.

Arguments are annotated as float, and return type is annotated as float.
"""
return weight / (height ** 2)
# Annotations can be accessed via the __annotations__ attribute
print(calculate_bmi.__annotations__)
```

```
# Output: {'weight': float, 'height': float, 'return': float}

# Annotations are hints and don't enforce type checking by default
result = calculate_bmi(70, 1.75)  # Works fine
result = calculate_bmi("70", "1.75")  # Also works, but might not be intended
```

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

Ans. Recursive functions: Recursive functions are functions that call themselves to solve a problem by breaking it down into smaller subproblems.

```
def factorial(n: int) -> int:
    """Calculate factorial using recursion."""
    # Base case
    if n <= 1:
        return 1
    # Recursive case
    return n * factorial(n - 1)

# Example usage
print(factorial(5))  # Output: 120 (5 * 4 * 3 * 2 * 1)

# Breaking down the recursion:
# factorial(5) = 5 * factorial(4)
#              = 5 * (4 * factorial(3))
#              = 5 * (4 * (3 * factorial(2)))
#              = 5 * (4 * (3 * (2 * factorial(1))))
#              = 5 * (4 * (3 * (2 * 1)))
#              = 120
```

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

Ans. General design guidelines for coding functions:

- Keep functions focused on a single task (Single Responsibility Principle)

- Use clear, descriptive names
- Keep functions reasonably short
- Use docstrings for documentation
- Follow DRY (Don't Repeat Yourself) principle
- Use default parameters appropriately
- Consider using type hints
- Handle errors gracefully
- Write functions that are easy to test

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

Ans.

- **Return values**

```
def add(a, b):
    return a + b
```

- **Modifying mutable arguments**

```
def append_item(lst, item):
    lst.append(item)
```

- **Global variables (though generally discouraged)**

```
global_result = 0
def update_global():
    global global_result
    global_result += 1
```

- **Raising exceptions**

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

- **Yielding values (for generators)**

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1
```