UNIT II: Functions and Strings

Subtopic 3: Function Composition and Recursion

Function Composition

3.1 Combining Multiple Functions to Solve Complex Problems

Function composition involves creating more complex functions by combining simpler ones. This approach allows you to break down complex tasks into smaller, manageable functions that can be reused and composed together.

Example:

Consider a scenario where you need to calculate the area of a rectangle and then use that area to determine how much paint is needed to cover it.

```python
def calculate_area(length, width):
    return length * width

def calculate_paint_needed(area, coverage_per_liter):
    return area / coverage_per_liter

length = 10
width = 5
coverage_per_liter = 8  # One liter covers 8 square meters

area = calculate_area(length, width)
paint_needed = calculate_paint_needed(area, coverage_per_liter)

print(f"Area of the rectangle: {area} square meters")
print(f"Paint needed: {paint_needed} liters")
# Output:
# Area of the rectangle: 50 square meters
# Paint needed: 6.25 liters


# Example: Function Composition
def calculate_area(length, width):
    return length * width

def calculate_paint_needed(area, coverage_per_liter):
    return area / coverage_per_liter

length = 10
width = 5
coverage_per_liter = 8  # One liter covers 8 square meters

area = calculate_area(length, width)
paint_needed = calculate_paint_needed(area, coverage_per_liter)
```

```python
print(f"Area of the rectangle: {area} square meters")
print(f"Paint needed: {paint_needed} liters")
# Output:
# Area of the rectangle: 50 square meters
# Paint needed: 6.25 liters
```

Recursion

3.2 Understanding Recursive Functions

A recursive function is a function that calls itself to solve a smaller instance of the same problem. Recursive functions typically have a base case (the condition under which the function stops calling itself) and a recursive case (the part where the function calls itself).

Example: Calculating Factorial

The factorial of a number `n` is the product of all positive integers less than or equal to `n`. It is denoted as `n!`.

Recursive Definition:

- `0! = 1` (base case)
- `n! = n * (n-1)!` (recursive case)

Example:

```python
def factorial(n):
    if n == 0:
        return 1  # Base case
    else:
        return n * factorial(n-1)  # Recursive case

result = factorial(5)
print(f"Factorial of 5: {result}")
# Output: Factorial of 5: 120


# Example: Calculating Factorial
def factorial(n):
    if n == 0:
        return 1  # Base case
    else:
        return n * factorial(n-1)  # Recursive case

result = factorial(5)
print(f"Factorial of 5: {result}")
# Output: Factorial of 5: 120
```

3.3 Example: Fibonacci Series

The Fibonacci series is a sequence where each number is the sum of the two preceding ones, usually starting with 0 and 1.

Recursive Definition:

- `F(0) = 0` (base case)
- `F(1) = 1` (base case)
- `F(n) = F(n-1) + F(n-2)` (recursive case)

Example:

```
def fibonacci(n):
    if n <= 0:
        return 0  # Base case
    elif n == 1:
        return 1  # Base case
    else:
        return fibonacci(n-1) + fibonacci(n-2)  # Recursive case

result = fibonacci(6)
print(f"Fibonacci number at position 6: {result}")
# Output: Fibonacci number at position 6: 8


# Example: Fibonacci Series
def fibonacci(n):
    if n <= 0:
        return 0  # Base case
    elif n == 1:
        return 1  # Base case
    else:
        return fibonacci(n-1) + fibonacci(n-2)  # Recursive case

result = fibonacci(6)
print(f"Fibonacci number at position 6: {result}")
# Output: Fibonacci number at position 6: 8
```

3.4 Practical Applications of Recursion

Example: Sum of a List

You can use recursion to calculate the sum of a list of numbers.

Example:

```
def sum_list(numbers):
    if not numbers:
        return 0  # Base case
    else:
        return numbers[0] + sum_list(numbers[1:])  # Recursive case
```

```python
numbers = [1, 2, 3, 4, 5]
result = sum_list(numbers)
print(f"Sum of the list: {result}")
# Output: Sum of the list: 15


# Example: Sum of a List
def sum_list(numbers):
    if not numbers:
        return 0  # Base case
    else:
        return numbers[0] + sum_list(numbers[1:])  # Recursive case

numbers = [1, 2, 3, 4, 5]
result = sum_list(numbers)
print(f"Sum of the list: {result}")
# Output: Sum of the list: 15
```

Example: Checking Palindrome

A recursive function can check if a string is a palindrome (reads the same forward and backward).

Example:

```python
def is_palindrome(s):
    if len(s) <= 1:
        return True  # Base case
    elif s[0] != s[-1]:
        return False
    else:
        return is_palindrome(s[1:-1])  # Recursive case

word = "radar"
result = is_palindrome(word)
print(f"Is '{word}' a palindrome? {result}")
# Output: Is 'radar' a palindrome? True


# Example: Checking Palindrome
def is_palindrome(s):
    if len(s) <= 1:
        return True  # Base case
    elif s[0] != s[-1]:
        return False
    else:
        return is_palindrome(s[1:-1])  # Recursive case

word = "radar"
```

```python
result = is_palindrome(word)
print(f"Is '{word}' a palindrome? {result}")
# Output: Is 'radar' a palindrome? True
```

Summary

Function composition and recursion are powerful techniques in Python that allow you to solve complex problems by breaking them down into smaller, manageable tasks. By mastering these techniques, you can write more efficient and elegant code to address a wide range of programming challenges.