**Module 13. Recursion**

**Learning Objectives**

* Understand the concept and structure of a recursive function.
* Identify the base case and recursive case in a recursive function.
* Implement recursive functions to solve problems like summing a list, factorials, Fibonacci numbers, and GCD.
* Analyze the benefits and drawbacks of using recursion in programming.
* Recognize potential performance issues and avoid stack overflow errors in recursive algorithms.

1. **Introduction**

In a program, a function can invoke another function, which in turn might call yet another function. Additionally, a function has the capability to call itself. Such a function is referred to as a recursive function. For instance, consider the function illustrated in the following program.

(Bad Recursion Example)

# This program has a recursive function.

# NOTE: This program causes infinite recursion

def main():

    print\_message()

def print\_message():

    print('Hi. Do you know what a recursive function is?')

    print\_message()

# Call the main function.

if \_\_name\_\_ == '\_\_main\_\_':

    main()

The output will be:

Hi. Do you know what a recursive function is?

Hi. Do you know what a recursive function is?

Hi. Do you know what a recursive function is?

… will be repeated infinitely.

In the above example, the ***print\_message*** function outputs the string 'Hi. Do you know what a recursive function is?' and then invokes itself again and again. There is no mechanism to terminate the recursive calls. This function acts like an infinite loop because there is no code to halt its repetition. If you execute this program, you will need to press Ctrl+C on the keyboard to stop its execution.

Similar to a loop, a recursive function needs a mechanism to control the number of times it executes. The following code is a revised version of the ***print\_message*** function. In this version, the ***print\_message*** function takes an argument that determines how many times the message should be displayed.

(Revised Version)

# This program demonstrates a simple recursive function to print a message a specified number of times.

def main():

    times = 4  # Number of times to print the message

    print\_message(times)

def print\_message(times):

    if times <= 0:  # Base case: When times is 0 or less, stop the recursion

        return

    else:

        print('Hi. Do you know what a recursive function is?')

        print\_message(times - 1)  # Recursive case: Call the function with times - 1

# Call the main function.

if \_\_name\_\_ == '\_\_main\_\_':

    main()

(Explanation **)**

* **Main Function:** The ***main*** function initializes the number of times the message should be printed and calls the ***print\_message*** function with that number.
* **Print Message Function:**
  + **Base Case:** When ***times*** is 0 or less, the function stops calling itself, which prevents infinite recursion.
  + **Recursive Case:** The function prints the message and then calls itself with times - 1, reducing the number of remaining prints by one each time.

**Key Concepts**

* **Base Case:** The condition under which the recursive function stops calling itself. This is crucial to prevent infinite recursion and stack overflow errors.
* **Recursive Case:** The part of the function where it calls itself with a modified argument, moving towards the base case.

1. **Copy Yourself to Do the Rest**

Recursion is a powerful technique in programming where a function calls itself to solve smaller instances of the same problem. This approach is particularly useful for problems that exhibit self-similarity, meaning the problem can be broken down into smaller, similar sub-problems.

**Example 1: Summing a List of Numbers**

Consider the task of summing a list of numbers using recursion. Here is a step-by-step breakdown of how the recursive function ***recsum*** works:

1. **Base Case:** The simplest instance of the problem, where no further recursion is needed. In this case, when the index reaches the length of the list, there are no more elements to sum, so the function returns 0.
2. **Recursive Case:** The function performs a small part of the work (adding the current element to the sum of the remaining elements) and then calls itself with a smaller instance of the problem (the next index in the list).

def recsum(number\_list, index):

    if index == len(number\_list):

        return 0

    return number\_list[index] + recsum(number\_list, index + 1)

num\_list = [4, 6, 3]

print(f"The sum of the numbers in {num\_list}: {recsum(num\_list, 0)}")

The output will be:

The sum of the numbers in [4, 6, 3]: 13

**Example 2: Factorial Calculation**

The factorial of a non-negative integer ***num*** is the product of all positive integers less than or equal to ***num***. The recursive definition of factorial is:

* **Base Case:** 0! = 1
* **Recursive Case:** num! = num × (num−1)!

Here is the recursive function to calculate factorial:

def factorial(num):

    if num == 0:

        return 1

    return num \* factorial(num - 1)

num = 5

print(f"The factorial of {num} is {factorial(num)}")

The output will be:

The factorial of 5 is 120

**Visualizing the Recursive Process**

To better understand how recursion works, let's trace the execution of ***factorial(3)***:

* ***factorial(3)*** calls ***factorial(2)***
* ***factorial(2)*** calls ***factorial(1)***
* ***factorial(1)*** calls ***factorial(0)***
* ***factorial(0)*** returns ***1*** (i.e., base case)
* ***factorial(1)*** returns 1 × 1 = 11
* ***factorial(2)*** returns 2 × 1 = 22
* ***factorial(3)*** returns 3 × 2 = 63

**Example 3: Fibonacci Numbers**

The Fibonacci sequence is a classic example to illustrate recursion. The sequence starts with 0 and 1, and each subsequent number is the sum of the two preceding ones.

Mathematically, it is defined as:

* **Base Cases:** When n is 0, the function returns 0. When n is 1, the function returns 1.
* **Recursive Case:** The function returns the sum of the Fibonacci numbers of n-1 and n-2, breaking the problem into smaller sub-problems until reaching the base cases.

# This program uses recursion to print numbers from the Fibonacci series.

def main():

    # Display the first 10 numbers in the Fibonacci series

    print("The first 10 numbers in the Fibonacci series are:")

    for number in range(10):

        print(fib(number))

# The fib function returns the nth number in the Fibonacci series.

def fib(n):

    if n == 0:

        return 0  # Base case: The 0th Fibonacci number is 0

    elif n == 1:

        return 1  # Base case: The 1st Fibonacci number is 1

    else:

        return fib(n - 1) + fib(n - 2)  # Recursive case: Sum of the two preceding numbers

# Call the main function.

if \_\_name\_\_ == '\_\_main\_\_':

    main()

(Explanation)

* **Main Function:**
  + The main function is the entry point of the program. It prints a message and then uses a for loop to print the first 10 Fibonacci numbers by calling the fib function.
* **Fib Function:**
  + **Base Cases:**
    - When ***n*** is 0, the function returns 0.
    - When ***n*** is 1, the function returns 1.
  + **Recursive Case:**
    - For any ***n*** greater than 1, the function returns the sum of ***fib(n-1)*** and ***fib(n-2)***. This is where the function calls itself with smaller values of ***n***.

**Example 4: Greatest Common Divisor (GCD)**

To further illustrate recursion, let's add an example of calculating the greatest common divisor (GCD) using the Euclidean algorithm.

# This program uses recursion to find the GCD of two numbers.

def main():

    # Get two numbers from the user

    num1 = int(input("Enter an integer: "))

    num2 = int(input("Enter another integer: "))

    # Display the GCD

    print(f"The greatest common divisor of {num1} and {num2} is {gcd(num1, num2)}")

# The gcd function returns the greatest common divisor of two numbers.

def gcd(x, y):

    if y == 0:

        return x  # Base case: If y is 0, x is the GCD

    else:

        return gcd(y, x % y)  # Recursive case: Call gcd with (y, x % y)

# Call the main function.

if \_\_name\_\_ == '\_\_main\_\_':

    main()

(Explanation)

* **Main Function:**
  + The main function prompts the user to enter two integers and then calls the ***gcd*** function to calculate and print their GCD.
* **GCD Function:**
  + **Base Case**: If ***y*** is 0, the function returns x as the GCD.
  + **Recursive Case**: The function calls itself with ***y*** and ***x % y*** (the remainder when ***x*** is divided by ***y***), gradually reducing the problem size until the base case is reached.

1. **Benefits and Drawbacks of Recursion**

Recursion offers clarity and elegance by allowing problems to be defined and solved in a self-referential manner, making the code more intuitive and concise. However, it can be less efficient due to the overhead of multiple function calls and risks stack overflow errors in deep recursion scenarios.

**Benefits:**

* **Clarity:** Recursive solutions can be more intuitive and easier to understand for problems that naturally fit a recursive structure.
  + For example, problems like traversing a tree, solving the Towers of Hanoi, or generating permutations can be more straightforwardly implemented using recursion.
* **Elegance:** Recursive code can be more concise and closely match the problem's definition.
  + Recursion allows a problem to be defined in a self-referential way, which can lead to cleaner and more readable code. For instance, the mathematical definition of the Fibonacci sequence translates directly into a simple recursive function.

**Drawbacks:**

* **Performance:** Recursive functions can be less efficient due to the overhead of multiple function calls.
  + Each recursive call adds a layer to the call stack, which consumes memory and processing time. In cases of deep recursion, this can lead to performance bottlenecks.
* **Stack Overflow:** Deep recursion can lead to stack overflow errors if the base case is not reached within a reasonable number of calls.
  + If the recursion is too deep or the base case is not defined properly, the program may exhaust the call stack's memory, leading to a stack overflow error. This is a common issue in naive implementations of recursive algorithms.

To summarize, recursion is a powerful technique for solving problems that can be broken down into smaller, similar sub-problems. While it offers clarity and elegance, it must be used judiciously to avoid performance issues and stack overflow errors. Understanding the base case and the recursive case is crucial for writing effective recursive functions.

**Summary**

1. A recursive function is a function that calls itself to solve smaller instances of the same problem.
2. An example of bad recursion is a function that calls itself indefinitely without a termination condition, leading to infinite recursion.
3. A revised recursive function includes a base case to stop the recursion and prevent infinite loops.
4. The base case in a recursive function is the condition under which the function stops calling itself.
5. The recursive case in a function is where the function performs part of the work and calls itself with modified arguments.
6. Recursion is useful for problems with self-similarity, such as summing a list, calculating factorials, and generating Fibonacci numbers.
7. The recursive process involves the function breaking down a problem into smaller sub-problems until it reaches the base case.
8. An example of recursion is the Euclidean algorithm for finding the greatest common divisor (GCD) of two numbers.
9. Recursion offers clarity and elegance in code but can be less efficient due to the overhead of multiple function calls.
10. While recursion simplifies complex problems, it must be used carefully to avoid performance issues and stack overflow errors.

**Programming Exercises**

1. **Recursive Power Calculation**

Design a recursive function that accepts two arguments, ***base*** and ***exponent***. The function should return the value of ***base*** raised to the power of ***exponent***. For example, ***power(3, 4)*** should return ***81***.

1. **Recursive String Reversal**

Design a recursive function that accepts a string as an argument and returns the reverse of the string.

For example, if the input is "recursion", the function should return "noisrucer".

1. **Recursive Palindrome Check**

Design a recursive function that accepts a string as an argument and returns ***True*** if the string is a palindrome, and ***False*** otherwise. A palindrome is a string that reads the same backward as forward. For example, "racecar" is a palindrome.

1. **Recursive Binary Search**

Design a recursive function that accepts a sorted list of integers and a target value. The function should return the index of the target value in the list, or ***-1*** if the target is not found. For example, in the list ***[1, 2, 3, 4, 5]***, the target value ***3*** should return index ***2***.

1. **Recursive Decimal to Binary Conversion**

Design a recursive function that accepts a non-negative integer and returns its binary representation as a string.

For example, if the input is 10, the function should return "1010".