python

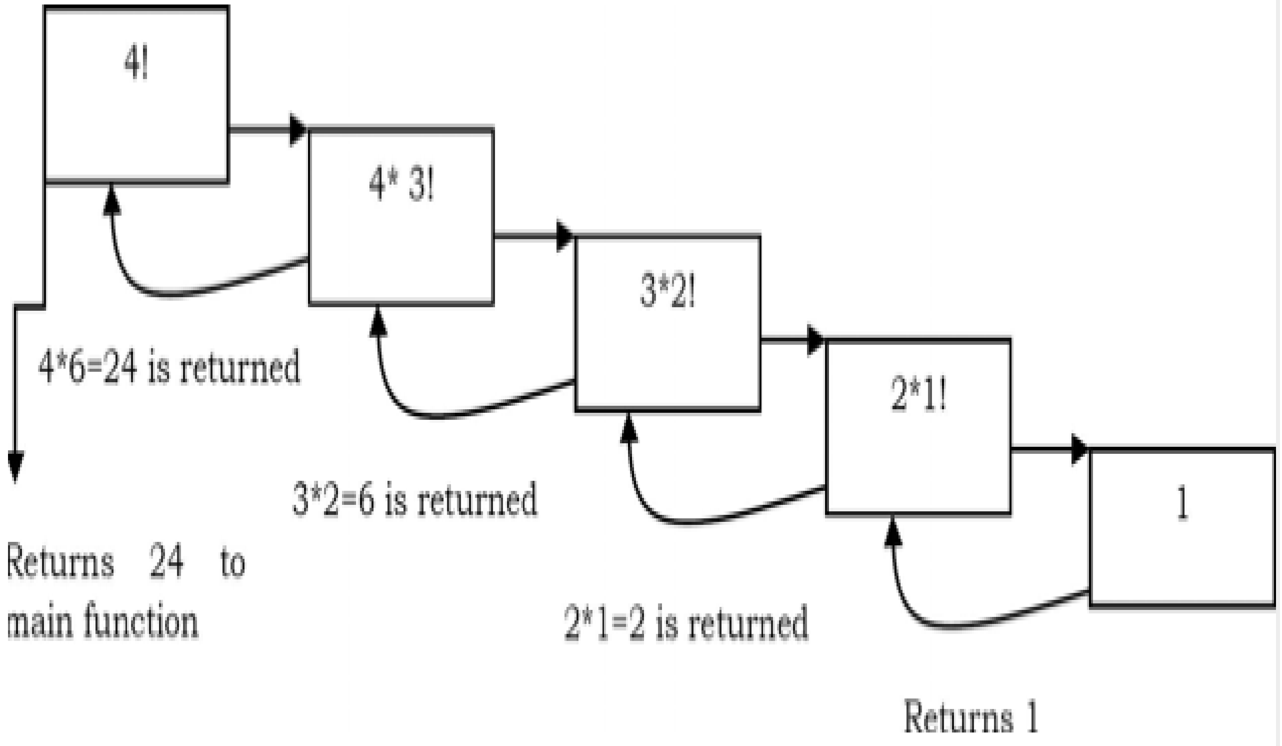
Course Id :INT 213

Recursion

- Recursion is a useful technique borrowed from mathematics.
- Recursive code is generally shorter and easier to write than iterative code. Generally, loops are turned into recursive functions when they are compiled or interpreted.
- Recursion is most useful for tasks that can be defined in terms of similar subtasks. For example, sort, search, and traversal problems often have simple recursive solutions.

Recursion

```
// calculates factorial of a positive integer
def factorial(n):
  if n == 0: return 1
  return n*factorial(n-1)
```



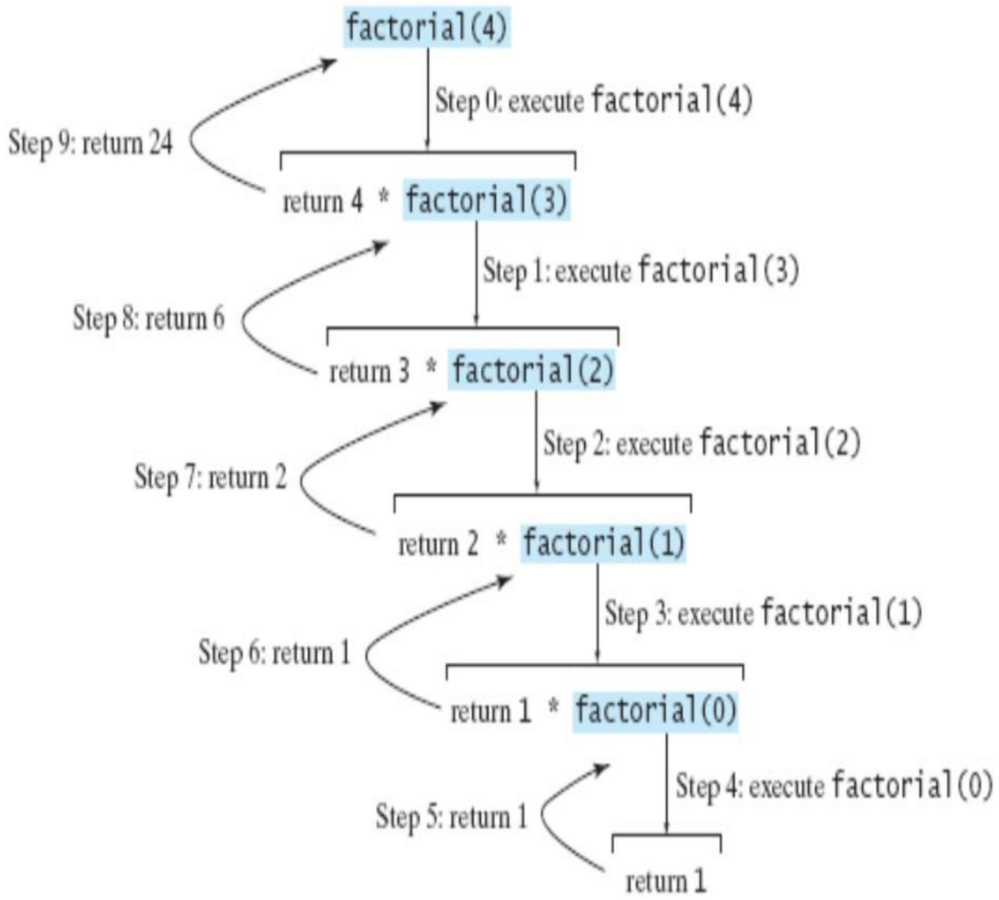
How program executed using recursion?

Program that prompts the user to enter a nonnegative integer and displays the factorial for the number is given below:

```
def main ():
    n = eval (input ("Enter a nonnegative integer: "))
    print ("Factorial of", n, "is", factorial (n))

# Return the factorial for the specified number
def factorial (n):
    if n == 0: # Base case
        return 1
    else:
        return n * factorial (n - 1)    # Recursive call

main ()         # Call the main function
```



Tail Recursion

- Tail recursion is a special case of recursion in which the last operation of a function is a recursive call.
- In tail recursive function, there are no pending operations to be performed on return from a recursive call.

Normal Recursion Vs Tail Recursion (Factorial of a number)

Normal Recursion

rec.py - C:\Users\Dipen\AppData\Local\Programs\Python\Python37-32\rec.py (3.7.4)

File Edit Format Run Options Window Help

```python
def fact(n):
    if n==1:
        return 1
    else:
        return n*fact(n-1)
def main():
    num = eval(input("enter the number"))
    x=fact(num)
    print(x)
main()
```

Tail Recursion

tr.py - C:\Users\Dipen\AppData\Local\Programs\Python\Python37-32\tr.py (3.7.4)

File Edit Format Run Options Window Help

```python
def fact(n,result):
    if n==1:
        return result
    else:
        return fact(n-1,n*result)
def main():
    num = eval(input("enter the number"))
    x=fact(num,1)
    print(x)
main()
```

How to convert a non tail recursive function into a tail recursive function?

- A non tail recursion function can be converted to a tail recursive function by adding one or more auxiliary parameters .
- For example, result is added as an auxiliary parameter in the definition of function **fact** in the previous example**.**

Fibonacci series:0 1 1 2 3 5 8 13 21 ……….

Enter the term number(for example term number of 0 is 1,2 is 4…….) in output we will have the number at that location

fibrec.py - C:/Users/Dipen/AppData/Local/Programs/Python/Python37-32/fibrec.py (3.7.4)

File Edit Format Run Options Window Help

```
def fib_norm(n1):
    if n1==1:
        return 0
    elif n1==2:
        return 1
    else:
        return fib_norm(n1-1)+fib_norm(n1-2)


def main():
    n= eval(input("enter the term no."))
    term = fib_norm(n)
    print("Fibonnaci term is",term)
main()
```

Normal Recursion Vs Tail Recursion (Fibonacci function)

fibrec.py - C:/Users/Dipen/AppData/Local/Programs/Python/Python37-32/fibrec.py (3.7.4)

File Edit Format Run Options Window Help

```python
def fib_norm(n1):
    if n1==1:
        return 0
    elif n1==2:
        return 1
    else:
        return fib_norm(n1-1)+fib_norm(n1-2)


def main():
    n= eval(input("enter the term no."))
    term = fib_norm(n)
    print("Fibonnaci term is",term)
main()
```

fibtailrec.py - C:/Users/Dipen/AppData/Local/Programs/Python/Python37-32/fibtailrec.py (3.7.4)

File Edit Format Run Options Window Help

```python
def fib_tail(n1,next,result):
    if n1==1:
        return result
    else:
        return fib_tail(n1-1, next+result,next)


def main():
    n= eval(input("enter the term no."))
    term = fib_tail(n,1,0)
    print("Fibonnaci term is",term)
main()
```

Recursion vs iteration

A recursive function generally takes more time to execute than an equivalent iterative approach. This is because the multiple function calls are relatively time-consuming. In contrast, while and for loops execute very efficiently. Thus, when a problem can be solved both recursively and iteratively with similar programming effort, it is generally best to use an iterative approach.

Recursion	Iteration
• Terminates when a base case is reached.	• Terminates when a condition is proven to be false.
• Each recursive call requires extra space on the stack frame(memory)	• Each iteration does not require extra space.
• If we get infinite recursion, the program may run out of memory and gives stack overflow.	• An infinite loop could forever since there is no extra memory being created.
• Solutions to some problems are easier to formulate recursively.	• Iterative solutions to a problem may not always be as obvious as a recursive solution.

Important points about Recursion

- Recursive algorithms have two types of cases, recursive cases and base cases.
- Every recursive function case must terminate at a base case.
- Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
- A recursive algorithm can be implemented without recursive function calls using a stack, but it’s usually more trouble than its worth. That means any problem that can be solved recursively can also be solved iteratively.
- For some problems, there are no obvious iterative algorithms.
- Some problems are best suited for recursive solutions while others are not.

Example Algorithms of Recursion

- Fibonacci Series, Factorial Finding
- Merge Sort, Quick Sort
- Binary Search
- Tree Traversals and many Tree Problems: InOrder, PreOrder PostOrder
- Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
- Dynamic Programming Examples
- Divide and Conquer Algorithms
- Towers of Hanoi
- Backtracking