Recursion

Lecture 8

Centre for Data Science, ITER
Siksha 'O' Anusandhan (Deemed to be University),
Bhubaneswar, Odisha, India.

Table Of Contents:

Introduction

- Recursion is the process of describing the computation in a function in terms of function itself.
- Recursive function comprises of two parts:
  1. Base part:- The solution of the simplest version of the problem, often termed stopping or termination criterion.
  2. Inductive part:- The recursive part of the problem that reduces the complexity of the problem by redefining the problem in terms of simpler version(s) of the original problem itself.

Recursion on Numbers(Example I)

- Recursion can be used to find the factorial of a given number
- Following algorithm can be used:

$$n! = \begin{cases} 1 & \text{if} \quad n == 0 \quad \text{or} \quad n == 1 \quad \text{(Base part)} \\ n * (n-1)! & \text{if } n > 1 \quad \text{(Inductive part)} \end{cases}$$

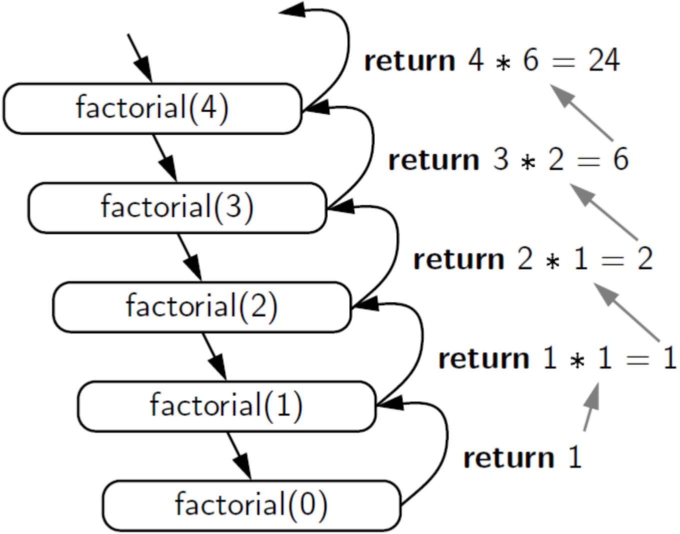
Recursion on Numbers(Example I Continued)

Following program finds the factorial of a given number:

```
def factorial(n):
    '''
    Objective: To compute factorial of a positive integer
    Input parameter: n-numeric value
    Return value: factorial of n- numeric value
    '''

    assert n>= 0
    if n==0 or n==1:
        return 1
    else:
        return n*factorial(n-1)

def main():
    '''
    Objective: To compute factorial of a number provided by the user
    Input parameter: None
    Return value: None
    '''
    n = int(input('Enter the number:'))
    result = factorial(n)
    print('Factorial of',n,'is',result)

if __name__=='__main__':
    main()
```

Recursion on Numbers(Example I Continued)

The following is the recursive tree to get the fatorial of positive integer 4:



Recursion on Numbers(Example II)

- Sum of the first n natural numbers can be found recursively.
- Following algorithm can be used:

$$sum(n) = \begin{cases} 1 & \text{if} \quad n == 1 \ \ \text{(Base part)} \\ n + sum(n-1) & \text{if } n > 1 \ \ \text{(Inductive part)} \end{cases}$$

Recursion on Numbers(Example II Continued)

Following program finds the sum of n positive integers number:

```
def sum(n):
    if n==1:
        return 1
    else:
        return n+sum(n-1)

def main():
    n = int(input('Enter the number:'))
    print('Sum of first n natural numbers is', sum(n))

if __name__=='__main__':
    main()
```

Recursion on Numbers(Example III)

- Recursion can be used to find the reverse of a given number
- Following algorithm can be used:

rev = 0

$$reverse(n, rev) = \begin{cases} rev & \text{if} \quad n == 0 \ \ \text{(Base part)} \\ reverse(n//10, rev * 10 + r) & \text{if } n > 0 \ \ \text{(Inductive part)} \\ where, r = n\%10 & \end{cases}$$

Recursion on Numbers(Example III Continued)

Following program finds the reverse of a given number:

```
def reverse(n,rev):
    if n==0:
        return rev
    else:
        r = n%10
        rev = rev*10+r
        return reverse(n//10,rev)

def main():
    rev = 0
    n = int(input('Enter the number:'))
    print('reverse of',n,'is',reverse(n,rev))

if __name__=='__main__':
    main()
```

Recursion on Numbers(Example IV)

- GCD of two given numbers x and y can be obtained using recursion
- Following algorithm can be used:

$$GCD(x, y) = \begin{cases} x & \text{if} \quad y == 0 \quad \text{(Base part)} \\ GCD(y, x\%y) & otherwise \quad \text{(Inductive part).} \end{cases}$$

Recursion on Numbers(Example IV Continued)

Following program finds the GCD of two given numbers x and y:

```
def GCD(x,y):
    if y==0:
        return x
    else:
        r = x%y
        x = y
        y = r
        return GCD(x,y)

def main():
    x = int(input('Enter x:'))
    y = int(input('Enter y:'))
    print('GCD of ',x,'and ',y,'is ',GCD(x,y))

if __name__=='__main__':
    main()
```

Recursion on Numbers(Example V)

- nth term of the Fibonacci series can be obtained using binary recursion.
- Following algorithm can be used:

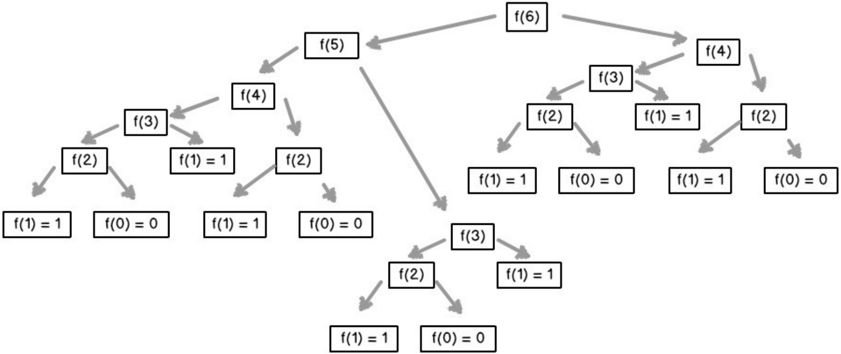
$$fib(n) = \begin{cases} 0 & \text{if} \quad n == 1 \;\; \text{(Base part)} \\ 1 & \text{if} \quad n == 2 \;\; \text{(Base part)} \\ fib(n-1) + fib(n-2) & \text{if } n > 2 \;\; \text{(Inductive part).} \end{cases}$$

Recursion on Numbers(Example V Continued)

Following program gives the nth term of the Fibonacci sequence:

```
def fib(n):
    if n==1:
        return 0
    elif n==2:
        return 1
    else:
        return fib(n-1)+fib(n-2)


def main():
    n = int(input('Enter a number: '))
    print('The fibonacci number at place ',n,'is',fib(n))

if __name__=='__main__':
    main()
```

The following is the recursive tree of the Fibonacci sequence:



Recursion on Strings(Example I)

- Recursion can be used to find the length of a string
- Following algorithm can be used:

$$\text{length(str1)} = \begin{cases} 0 & \text{if} \quad \text{str}== \text{''} \quad \text{(Base part)} \\ 1 + \text{length(str1[1:])} & \text{otherwise} \quad \text{(Inductive part).} \end{cases}$$

Recursion on Strings(Example I conitnued)

Following program gives the length of a given string:

```
def length(str1):
    '''
    Objective: To determine the length of input string
    Input parameter: str1- string
    Return value: numeric
    '''

    if str1=='':
        return 0
    else:
        return 1+length(str1[1:])

def main():
    '''
    Objective: To determine length of string
    Input parameter: None
    Return value: None
    '''
    str1 = input('Enter the string:')
    result = length(str1)
    print('Length of string',str1,'is',result)

if __name__=='__main__':
    main()
```

Recursion on Strings(Example II)

- Reverse of a string can be found recursively
- Following algorithm can be used:

$$\text{reverse(str1)} = \begin{cases} str1 & \text{if} \quad \text{str}== \text{''} \quad \text{(Base part)} \\ \text{str1[-1]}+\text{reverse(str[:-1])} & \text{otherwise} \quad \text{(Inductive part)}. \end{cases}$$

Recursion on Strings(Example II conitnued)

Following program finds the reverse of a given string:

```
def reverse(str1):
    if str1=='':
        return str1
    else:
        return str1[-1]+reverse(str1[:-1])


def main():
    str1 = input('Enter a string:')
    print('Reverse of',str1,'is',reverse(str1))

if __name__=='__main__':
    main()
```

Recursion on Strings(Example III)

- Recursion can be used to check whether a given string is a palindrome or not
- Following algorithm can be used:

$$\text{isPalindrome(str1)} = \begin{cases} \textit{True} & \text{if str== ''(Base part)} \\ \textit{False} & \text{if str1[0]!=str1[-1] ''(Base part)} \\ \text{isPalindrome(str1[1:-1])} & \text{otherwise} \quad \text{(Inductive part).} \end{cases}$$

Recursion on Strings(Example III continued)

Following program checks whether a given string is a palindrome or not

```
def isPalindrome(str1):
    if str1 == '':
        return True
    else:
        return(str1[0]==str1[-1] and isPalindrome(str1[1:-1]))

def main():
    str1 = input('Enter the string:')

    if isPalindrome(str1):
        print('String is a palindrome.')
    else:
        print('String is not a palindrome.')

if __name__=='__main__':
    main()
```

Recursion on Lists(Example I)

- Recursion can be used to find the sum of elements of a list.
- Following algorithm can be used:

$$\text{sumls(ls,n)} = \begin{cases} 0 & \text{if} \quad \text{n} == 0 \quad \text{(Base part)} \\ \text{ls[n-1]+sumls(ls,n-1)} & \text{otherwise} \quad \text{(Inductive part).} \end{cases}$$

Following program finds the sum of the elements of the given list:

```
def sumls(ls,n):
    if n==0:
        return 0
    else:
        return ls[n-1]+sumls(ls,n-1)

def main():
    ls = eval(input('Enter list elements:'))
    n = len(ls)
    print('Sum of elements of list is',sumls(ls,n))

if __name__=='__main__':
    main()
```

Recursion on Lists(Example II)

- Given a list of lists, the nesting of lists may occur up to any arbitrary level.
- Flattening of a list means to create a list of all data values in the given list.
- The data values in the flattened list appear in the left to right order of their appearance in the original list, ignoring the nested structure of the list.
- Thus the list [1,[2,[3,4]]] can be flattened to obtain [1,2,3,4].
- Following algorithm can be used:
  For every element in list 1
  if i is not a list
  append it to list 2
  otherwise flatten the list i

Recursion on Lists(Example II continued)

Following program finds the reverse:

```
def flatten(ls1,ls2=[]):
    for element in ls1:
        if type(element)!=list:
            ls2.append(element)
        else:
            flatten(element,ls2)
    return ls2

def main():
    ls1 = eval(input('Enter the list:'))
    result = flatten(ls1)
    print('Flattened List:',result)

if __name__=='__main__':
    main()
```

Shallow copy and Deep copy

- Simply assigning a list object to another name does not create a copy of the list; instead, both the names refer to the same list object.
- So when a change is made in any of the list it appears in both the lists.

Shallow copy and Deep copy

Shallow copy

- A shallow copy means constructing a new collection object and then populating it with references to the child objects found in the original.
- The copying process does not recurse and therefore won't create copies of the child objects themselves
- It means that any changes made to a copy of object do reflect in the original object.

Shallow copy and Deep copy

```python
## Making shallow copy
import copy

ls1 = [[1,2,3],[4,5,6],[7,8,9]]

ls2 = copy.copy(ls1)
print('ls1=',ls1)
print('ls2=',ls2)

ls2[0] = [0,0,0]
print('ls1=',ls1)
print('ls2=',ls2)

ls2[1][2] = 0
print('ls1=',ls1)
print('ls2=',ls2)
```

Deep copy

- A deep copy means first constructing a new collection object and then recursively populating it with copies of the child objects found in the original
- In case of deep copy, a copy of object is copied in other object.
- It means that any changes made to a copy of object do not reflect in the original object.

Shallow copy and Deep copy

```
## Making deep copy

import copy

ls1 = [[1,2,3],[4,5,6],[7,8,9]]

ls2 = copy.deepcopy(ls1)
print('ls1=',ls1)
print('ls2=',ls2)

ls2[0] = [0,0,0]
print('ls1=',ls1)
print('ls2=',ls2)

ls2[1][2] = 0
print('ls1=',ls1)
print('ls2=',ls2)
```

Conclusion

- Python allows a function to call itself, possibly with new parameter values. This technique is called recursion.
- Recursion allows us to break a large task down to smaller tasks by repeatedly calling itself.
- Every recursive function must have at least two cases: the inductive case and the base case.
- A recursive function requires a base case to stop execution, and the call to itself which gradually leads to the function to the base case.
- The inductive case is the more general case of the problem we are trying to solve.
- Recursion is not hard to implement in the right circumstances. It's important to make sure that the algorithm terminates, especially in cases where data corruption may occur.

References

[1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, Pearson Education India, Inc., 2017.