Functions – Recursion

INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY
DELHI

Functions – Recap

- Functions are defined with **`def <fn-name> (<parms>):`**
- A defined function can be called from main or another function
- Calling a function - arguments are passed, which are assigned to parameters - which can be used in function body
- Variables defined / used in a function, including parameters, are local variables (global vars can be used by explicitly declaring it)

Recursive Functions in Maths

- A recursive definition is one where the defined term is used in the definition
- Examples of it in class xii
  - Recursive definition for the factorial computation

    ```
    factorial(n) = n*factorial(n-1)

    factorial(0) = 1
    ```

  - Recursive definition for the Fibonacci Sequence (0,1,1,2,3,5,8,13, …)

    ```
    fibonacci(n) = fibonacci(n-1) + fibonacci(n-2) [for n>1]

    fibonacci(0) = 0

    fibonacci(1) = 1
    ```

- All recursive definitions must have:
  - Base Case: Which is known without recursion
  - Recursive part: where the problem is expressed as a smaller version of the problem itself.

Recursive algorithm

- A recursive algorithm uses itself to solve one or more smaller identical problems. For example
- Looking up a telephone number in a phone book :
  - Go to the middle page. If name is on the middle page stop.
  - Otherwise look in the first half or the second half depending on whether the name is alphabetically before or after the names on the middle page.
  - Repeat the process on this remaining portion and so on until you find the desired number
- Comparing two lists for equality
  - Compare the first item, if not same, lists not equal
  - Otherwise, compare the rest of the list
- Recursive functions like factorial, fibonacci etc, have their recursive algos given in the function definition

Recursion in Python functions

- Functions that call themselves are called recursive functions
  - Allowing recursive functions requires care in underlying implementation
  - Not all languages allowed recursive functions
  - Most modern programming languages allow it
- Python allows recursive functions, i.e. a function can call itself
- All recursive functions (like in any recursive definition) must have
  - A Base case : In which there is no recursive call to the function, and which has a clear return value
  - Recursive part: Where the function itself can be called but with different parameters
- Recursive call has to be such that eventually the base case is reached, so parameters are changed in the recursive call
- If the recursive part is such that the base case will not be reached - then we have infinite recursion (the program will never end)
- We must ensure that base case will *always* be reached

A Simple Example

```
def f(x):
    if x==0:     # base case
        return 0
    else:
        return 1+f(x-1)
```

- Trivial fn - counts from 0 till x, incr 1 at every call
- Recursive call: parm is x-1, i.e. reduced every time by 1
- Base case condition is x==0 - it will necessarily be reached eventually if x is +ve
- If x is -ve, then x will never reach zero - infinite recursion
- Can avoid by changing the base case condition to x<=0

Recursion Function Examples

```
def factorial(n):
  if n <= 1: #base case
    return 1
  else:
    return (n*factorial(n-1))
```

Note: recursive call with n-1, so parm will eventually reach n<=1 i.e. the base case

If n == 1 in base case, then infinite recursion if n is negative

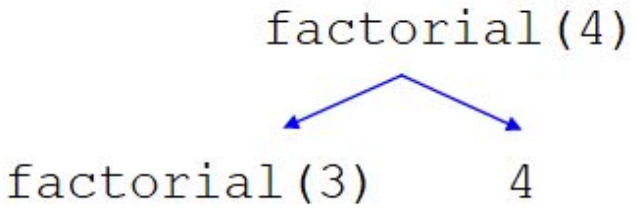
```
def digits(n):
  if n <= 9:      # base case
    return 1
  else:
    return (1 + digits(n//10))
```

Note recursive call with n//10 - means every time parm is n//10, so will eventually hit single digit

It will terminate for -ve nos also - but give a wrong answer

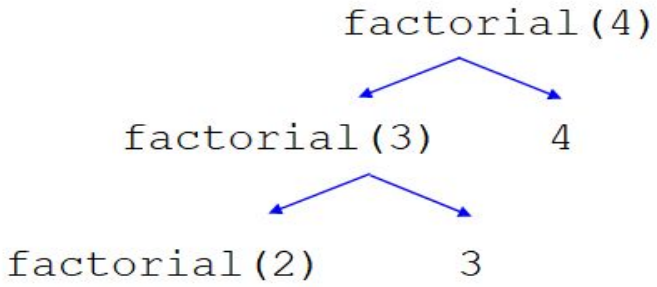
To change it, if n is -ve, make it +ve

Factorial Function Execution



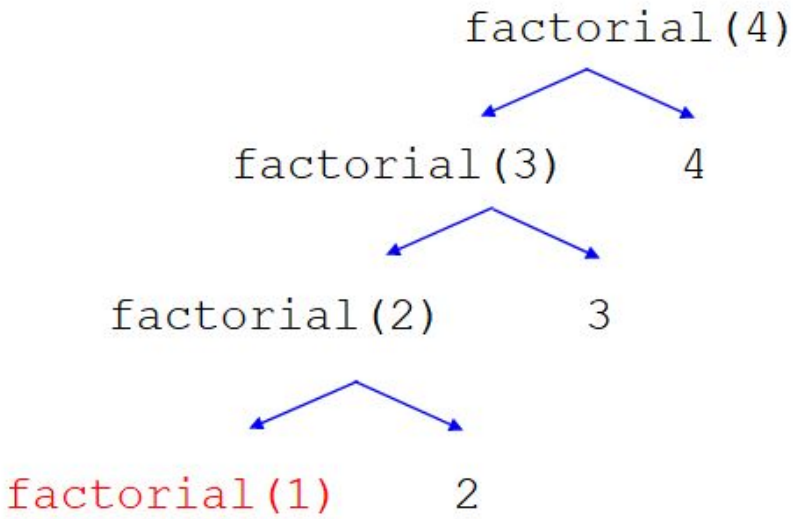

```
def factorial(n):
  if n <= 1:   #base case
    return 1
  else:
    return (n*factorial(n-1))
```

Factorial Function Execution



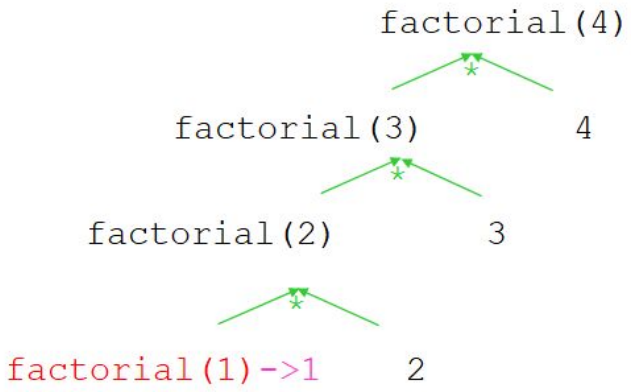

```
def factorial(n):
  if n <= 1:   #base case
    return 1
  else:
    return (n*factorial(n-1))
```

Factorial Function Execution



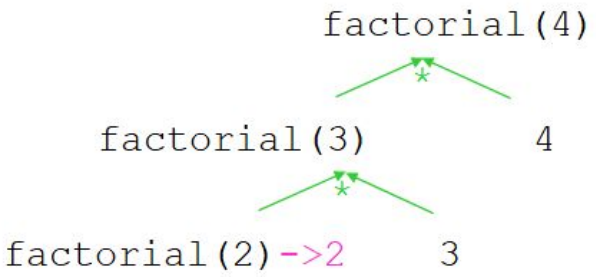

```
def factorial(n):
  if n <= 1:   #base case
    return 1
  else:
    return (n*factorial(n-1))
```

Factorial Function Execution



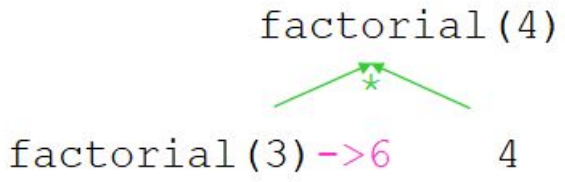

```
def factorial(n):
  if n <= 1:   #base case
    return 1
  else:
    return (n*factorial(n-1))
```

Factorial Function Execution




```
def factorial(n):
  if n <= 1:   #base case
    return 1
  else:
    return (n*factorial(n-1))
```

Factorial Function Execution




```
def factorial(n):
  if n <= 1:   #base case
    return 1
  else:
    return (n*factorial(n-1))
```

Factorial Function Execution

```
factorial(4)->24
```

```
def factorial(n):
  if n <= 1:   #base case
    return 1
  else:
    return (n*factorial(n-1))
```

Quiz – Text

Q) What should be the base case for the following code:

```
def is_palindrome(s):
    """
    This function checks whether a given string is a
    palindrome or not.
    A palindrome is a string that spells the same when read
    forwards or backwards
    """
    if ____________:
        return True
    else:
        return s[0] == s[-1] and is_palindrome(s[1:-1])
```

Quiz – Solution

Q) What should be the base case for the following code:

```
def is_palindrome(s):
    """
    This function checks whether a given string a palindrome
    or not.
    A palindrome is a string that spells the same when read
    forwards or backwards
    """
    if len(s) <= 1:
        return True
    else:
        return s[0] == s[-1] and is_palindrome(s[1:-1])
```

Execution of recursive functions

- When a function is called, a separate space (called frame) is allocated to this call where all local vars (incl parms) reside - function statements operate on these
- When a function returns (i.e. completes its execution) - frame is released and local vars disappear
- When a fn f1 calls another function f2:
  - frame for f1 remains (as f1 has not finished yet),
  - a new frame is allocated for f2 with its local vars
  - control jumps to f2 - and starts executing body of f2
  - On completing execution of f2, frame for f2 is freed,
  - control returns to f1 at the point where f2 was called
  - f1 resumes execution from there
- When f1 completes, its frame is released, control back to main

Execution of recursive functions

- Recursive call is similar to a function calling another function - a recursive call is treated as a regular fn call
- A new frame is created with local vars (incl parm) for each new call
- This (calls to functions) keep happening till the base case is hit
- Base case has no call - it returns value to the previous (i.e. last-but-one) call and its frame is released
- Now the previous call can complete, as the function call it had made has completed and control has returned - so it executes statements after the call, if any, and finishes - returns to caller
- The same happens now in caller … repeats till the first call
- Let us see the visualization in pythontutor - running factorial

Functions: Global and local var

- Global vars - those declared in the main scope i.e. outside of any fn - they can be accessed anywhere
- Local vars - those created in the function - do not have existence outside the function
- In a python function, variables only referenced are considered global unless a local var exists; if a variable is assigned a value, it is treated as local unless declared global
  - Allows use of global names of functions, etc (and global values)
  - Disallows accidental changing of a global variable
- If we have a var pointing to a mutable object, then assigning to an element is not changing the var, hence it is allowed
  - So, we can pass a list to a function, and have the fn change items
  - But we cannot pass a string/int - and have its value changed by the function (a function can return a new string/int)

Comparing recursive and iterative solution

Problem: Sum of 1st n natural no.s

Runtime for different values of n

n	Iterative Sol.	Recursive Sol.
100	8.845e-05	0.000326
1000	0.00019	0.0018
10000	0.00133	0.01156

```
#Iterative Solution
def sum_iterative(n):
  s = 0
  for i in range(1,n+1):
    s=s+i
  return s
```

```
#Recursive Solution
def sum_recursive(n):
  if n==1:
    return 1
  return n+sum_recursive(n-1)
```

Why Recursive Functions ?

- Recursive functions are not *necessary* - anything that can be done with recursion, can be done without it also
  - Some of the older languages did not have recursion
  - Most modern languages provide recursive functions
  - Recursive definitions are sometimes the most natural
- Many data structures (lists, trees, etc) are naturally recursive - for operations on them recursion is most natural
  - A binary tree: root node + l tree + r tree
  - A list: one item, followed by a list
- Recursive functions are often elegant, and compact

Quiz – Numeric

Q) How many times will the exponent function be called ?

```
def exponent(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return exponent(x, n // 2) * exponent(x, n // 2)
    else:
        return x * exponent(x, (n - 1))

exponent(10,3)
```

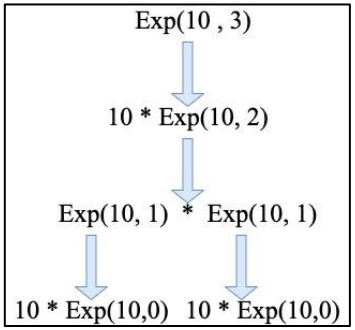
Quiz – Solution

Q) How many times will the exponent function be called ?

```
def exponent(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return exponent(x, n // 2) * exponent(x, n // 2)
    else:
        return x * exponent(x, (n - 1))

exponent(10,3)
```

Answer : 6



Care- avoid infinite Recursion

- As in while loop, we can have an infinite computation if not done carefully
- When defining a recursive function, must ensure:
  - There is a base case - in which no recursive call is made
  - Recursive calls are such that eventually the base class will be hit, ending the recursion
- Usually the parameter value of the recursive call will be different from the original call - and movement towards the direction that the base class will be reached
- All the frames created during recursive call take space in the memory and can lead to a Stack Overflow Error. To prevent this Python stops recursion at the 1000th recursive call.
  - This limit can be modified using the sys.setrecursionlimit(<limit>) function of the sys module. To be used with caution.

Care with Recursion – Fibonacci

- Fibonacci numbers are defined recursively
- fib(n): if n=0, return 0; if n==1, return 1; else fib(n-1)+fib(n-2)
- We can implement them directly in this manner also:

```
def fib(n):
    if n in {0, 1}:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

- But this implementation will be extremely inefficient - why?
  - E.g. for fib(5), it will compute fib(2) four times, fib(3) two times..
  - This is very inefficient - takes a lot of time (fib(2) computed 50+ times for n = 10!)
- How to make it recursive but not inefficient?

Fibonacci without duplicate recursion

- When a recursive fn is called with same value many times - we can save its value and use it - called memoization
- It is like creating your own cache of values computed
- If a recursive call is called with same params, it is not computed but looked up
- Cache best impl in python using dictionary
- For N=25, std fib ~0.5 sec, one on right about .01% of it
- For N=40, std fn - takes ~2mts, on right: .00007 sec!

```
cache = {0:0, 1:1}

def fib2(n):
  if n in cache:
    return cache[n]
  else:
    cache[n]=fib2(n-1)+fib2(n-2)
    return cache[n]
```

Quiz – Single Correct

Q) What would be the output for the following code:

A. 17
B. 5
C. 30
D. 18

```
def test(i,j):
    if(i==0):
        return j
    else:
        return test(i-1,i+j)
print(test(5,3))
```

Quiz – Solution

Q) What would be the output for the following code:

A. 17
B. 5
C. 30
D. 18

```
def test(i,j):
    if(i==0):
        return j
    else:
        return test(i-1,i+j)
print(test(5,3))
```

Explanation :

test(5,3) -> test(4,8) -> test(3,12) -> test(2,15) -> test(1,17) -> test(0,18) -> 18

Recursive Function Examples

```
def lenl(l):
    if l == []:
        return 0
    else:
        return 1 + lenl(l[1:])
===
def setl(s):
    if s == set():
        return 0
    else:
        x = s.pop()
        return 1 + setl(s)
```

```
def gcd(x,y):
    if (y == 0):
        return x
    else:
        return gcd(y, x%y)
====
def palin(s):
    if len(s)<=1:
        return True
    else:
        if s[0] == s[-1]:
            return palin(s[1:-1])
        else:
            return False
```

Quiz

Q) Complete the code such that the function returns the smallest element in the list of numbers.

```
def find_smallest(numbers):
    if len(numbers) == 1:
        return numbers[0]
    else:
        sm = find_smallest(numbers[1:])
        return ____________________
```

Quiz(Solution)

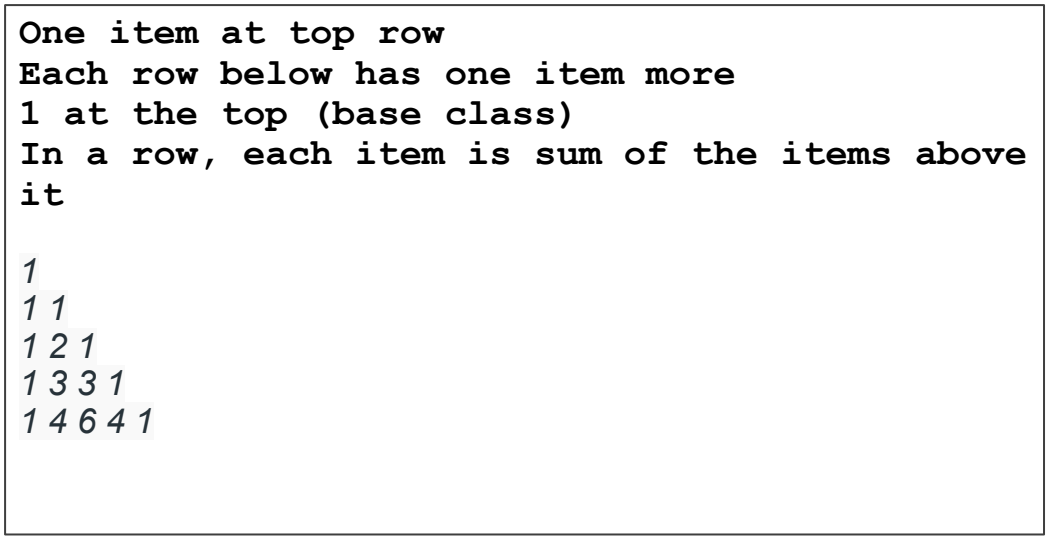
Q) Complete the code such that the function returns the smallest element in the list of numbers.

```
def find_smallest(numbers):
    if len(numbers) == 1:
        return numbers[0]
    else:
        sm = find_smallest(numbers[1:])
        return min(numbers[0], sm)
```

Exercise – Do it yourself

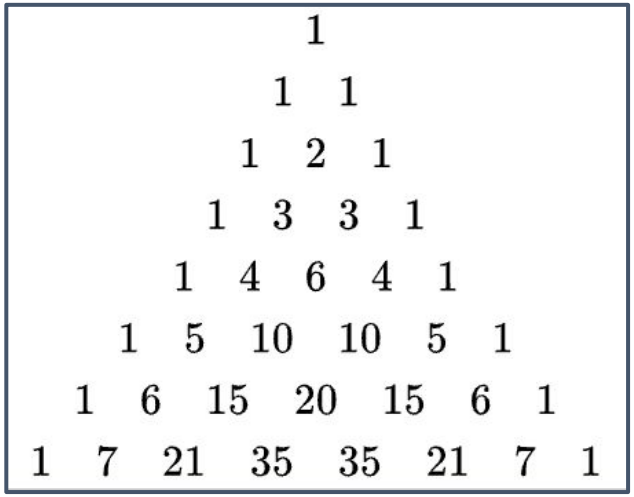
- Write a recursive function to sum all the elements of a list
- Write a recursive function to return the sum of the digits in a given integer

Example: Pascal's Triangle nth Row



```
One item at top row
Each row below has one item more
1 at the top (base class)
In a row, each item is sum of the items above
it

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```



1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

1 6 15 20 15 6 1

1 7 21 35 35 21 7 1

Print Pascal's Triangle – recursive, iterative

```
def pascal(n):
    if n == 1:
        print([1])
        return [1]
    else:
        line = [1]
        prev = pascal(n-1)
        for i in range(len(prev)-1):
            line.append(prev[i] + prev[i+1])
        line += [1]
    print (line)
    return line
```

```
def pascal(n):
    a = [1]
    print(a)
    for i in range(n+1):
        a.insert(0,0)
        a.insert(len(a)+1, 0)
        b = [a[i]+a[i+1] for i in
range(len(a)-1)]
        print(b)
        a = b.copy()
```

Pascal's Triangle – nth row

```
def pascal(n):
    if n == 1:
        return [1]
    else:
        line = [1]
        prev = pascal(n-1)
        for i in
range(len(prev)-1):
            line.append(prev[i] +
prev[i+1])
        line += [1]
    return line

print(pascal(4))
```

Try doing it with loops

Hard to implement this scheme

Have to recognize that each row is : NC0, NC1, ..., NCN - 1, NCN

Exercise

Printing of Pascal's triangle - will call for 2nd, 3rd, … rows again and again

Check how many times it calls the function

Change it while keeping recursion but calling 2nd, 3rd, etc only one time

Summary – Recursive Functions

- Functions can be recursive, i.e. a function calls itself. A recursive function must have
  - A base case where a value is returned
  - Recursive calls should move towards the base case
- Recursive functions are often natural (particularly for recursive data structures) and elegant, but they are generally not efficient as compared to iterative options
- Take care when using recursion - avoid infinite recursion, and avoid unnecessary extra calls