M110: Python Programming

Self-Study#2

Recursion

AOU
الجامعة العربية المفتوحة
Arab Open University
Lebanon

Prepared by Dr. Ahmad Mikati

Topics

- Introduction to Recursion
- Problem Solving with Recursion
- Examples of Recursive Algorithms

Introduction to Recursion

You have seen instances of functions calling other functions. In a program, the main function might call function A, which then might call function B.
It's also possible for a function to call itself.
A function that calls itself is known as a ***recursive function***

- **Recursive function**: a function that calls itself
- Recursive function must have a way to control the number of times it repeats
  - Usually involves an `if-else` statement which defines when the function should return a value and when it should call itself
- Depth of recursion: the number of times a function calls itself

Introduction to Recursion

Now, consider the below program that is intended to display a string in a recursive way.

```
# This program has a recursive function, badly implemented.
def main():
    message()
def message():
    print('This is a recursive function.')
    message()
# Call the main function.
main()
```

The message function displays the string 'This is a recursive function' and then calls itself.

Each time it calls itself, the cycle is repeated. Can you see a problem with it?

There's no way to stop the recursive calls. This function is like an indefinite loop because there is no code to stop it from repeating.

If you run this program, you will have to press Ctrl+C on the keyboard to interrupt its execution.

Introduction to Recursion

Like a loop, a recursive function must have some way to control the number of times it repeats.

The code in the below Program shows a modified version of the message function. In this program, the message function receives an argument that specifies the number of times the function should display the message.

```
# This program has a recursive function; correctly
implemented.
def main():
    message(5)
def message(times):
    if times>0:
        print('This is a recursive function.')
        message(times-1)
# Call the main function.
main()
```

The *message* function in this program contains an if statement that controls the repetition. If the ***times*** parameter is greater than zero, the message 'This is a recursive function' is displayed, and then the function calls itself again, but with a smaller argument.

Introduction to Recursion

The first time the function is called, the times parameter is set to 5. When the
function calls itself, a new instance of the times parameter is created, and the value 4 is passed into it. This cycle repeats until finally, zero is passed as an argument to the function. This is illustrated in Figure A.

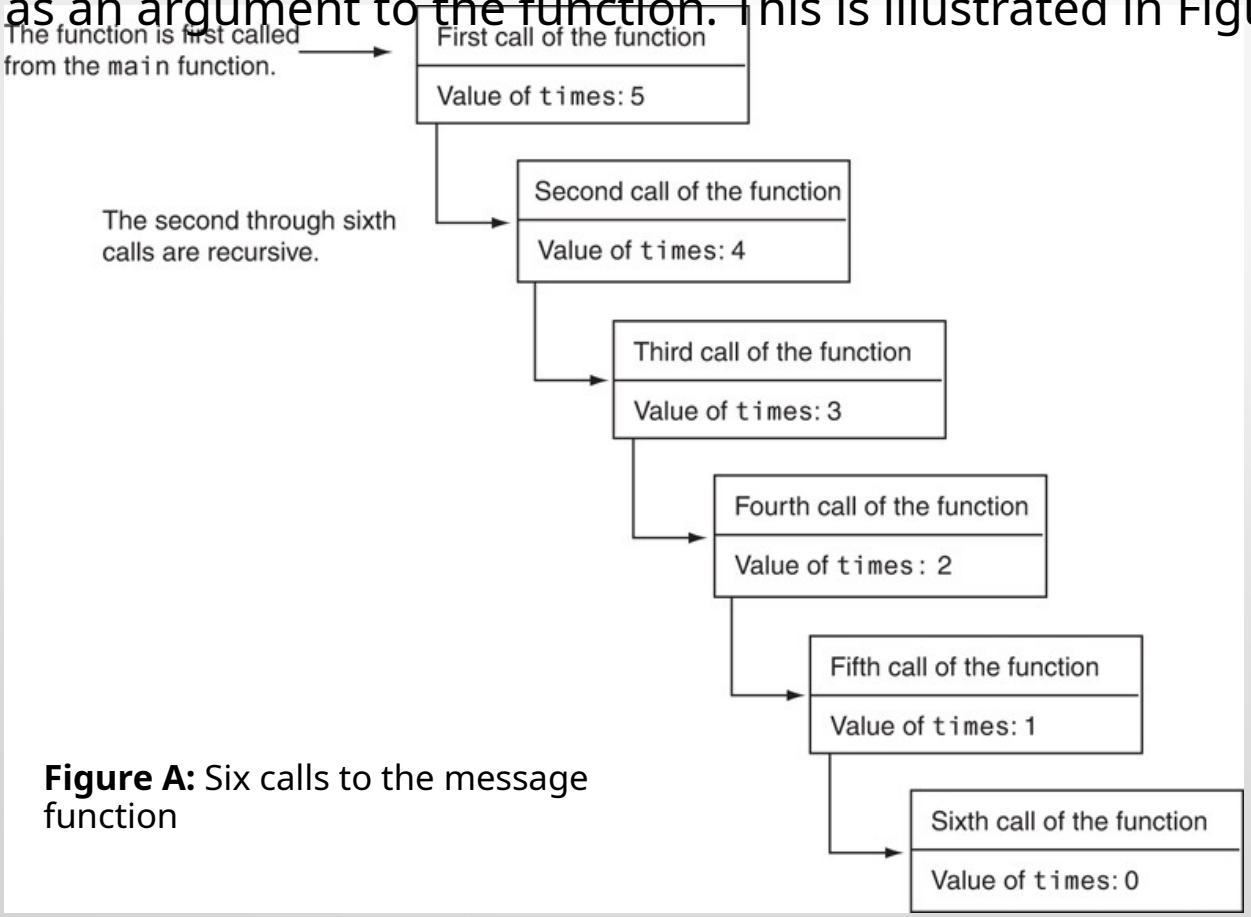


Figure A: Six calls to the message function

Introduction to Recursion

As you can see in the figure, the function is called six times. The first time it is called from the main function, and the other five times it calls itself.

The number of times that a function calls itself is known as the *depth of recursion*. In this example, the depth of recursion is five. When the function reaches its sixth call, the times parameter is set to 0. At that point, the if statement's conditional expression is false, so the function returns. Control of the program returns from the sixth instance of the function to the point in the fifth instance directly after the recursive function call. This is illustrated in Figure B.

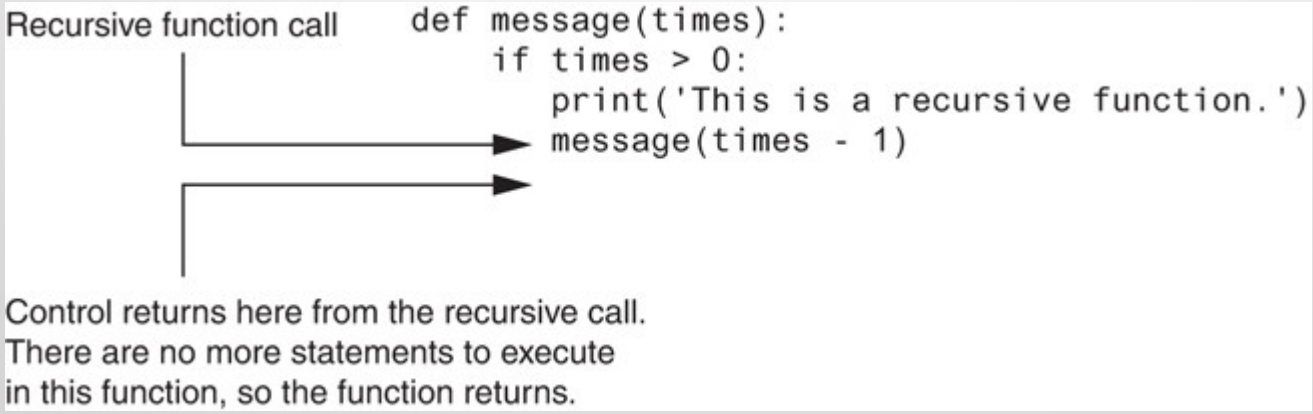


Figure B: Control returns to the point after the recursive function call

Problem Solving with Recursion

A problem can be solved with recursion if it can be broken down into smaller problems that are identical in structure to the overall problem.

- Recursion is a powerful tool for solving repetitive problems
- Recursion is not always required to solve a problem
  - Any problem that can be solved recursively can be solved with a loop
  - Recursive algorithms usually less efficient than iterative ones
    - Due to *overhead* of each function call

Problem Solving with Recursion

Some repetitive problems, however, are more easily solved with recursion than with a loop. Where a loop might result in faster execution time, the programmer might be able to design a recursive algorithm faster.

In general, a recursive function works as follows:

- If the problem can be solved now, without recursion, then the function solves it and returns.
- If the problem cannot be solved now, then the function reduces it to a smaller but similar problem and calls itself to solve the smaller problem

In order to apply this approach, first, we identify at least one case in which the problem can be solved without recursion. This is known as the ***base case***.

Second, we determine a way to solve the problem in all other circumstances using recursion. This is called the ***recursive case***.
In the recursive case, we must always reduce the problem to a smaller version of the original problem. By reducing the problem with each recursive call, the base case will eventually be reached, and the recursion will stop.

Recursive Programming- **Sum**

- Consider the problem of computing the sum of all the numbers between **1** and any positive integer **N**
- This problem can be recursively defined as:

$$\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i = N + N-1 + \sum_{i=1}^{N-2} i$$

$$= N + N-1 + N-2 + \sum_{i=1}^{N-3} i$$

$$\vdots$$

Recursive Programming- **Sum**

- Python's looping constructs make implementing this process easy.

$$sum(1) = 1$$
$$sum(N) = N + sum(N - 1) \text{ ; if } N > 1$$

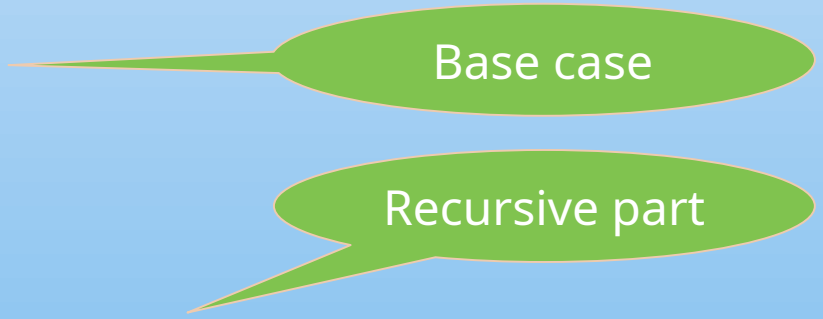
- Consider what happens when the definition is applied to the problem of calculating sum(4):

$$sum(4) = 4 + sum(3)$$
$$= 4 + 3 + sum(2)$$
$$= 4 + 3 + 2 + sum(1)$$
$$= 4 + 3 + 2 + 1$$

- The fact that sum(1) is defined to be 1 without making reference to further invocations of sum saves the process from going on forever and the definition from being circular.

Recursive Programming- **Sum**

```
# This program returns the sum of 1 to num
def sumAll(num):
    if num == 1:
        return 1
    else:
        return num + sumAll(num-1)
x=eval(input('Enter a positive number'))
print(sumAll(x))
```



Recursive Programming- **Factorial**

Let's take an example from mathematics to examine an application of recursive functions.

- In mathematics, the $n!$ notation represents the factorial of a nonnegative number *n. It* can be defined by the following rules:
  - For $n = 0$, $n! = 1$
  - For $n > 0$, $n! = 1 \times 2 \times 3 \times \ldots \times n$
- The above definition lends itself to recursive programming
  - $n = 0$ is the base case
  - $n > 0$ is the recursive case
    - $n! = n \times (n-1)!$

Recursive Programming- **Factorial**

- The *factorial* for any positive integer n, written n!, is defined to be the product of all integers between 1 and n inclusive.

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

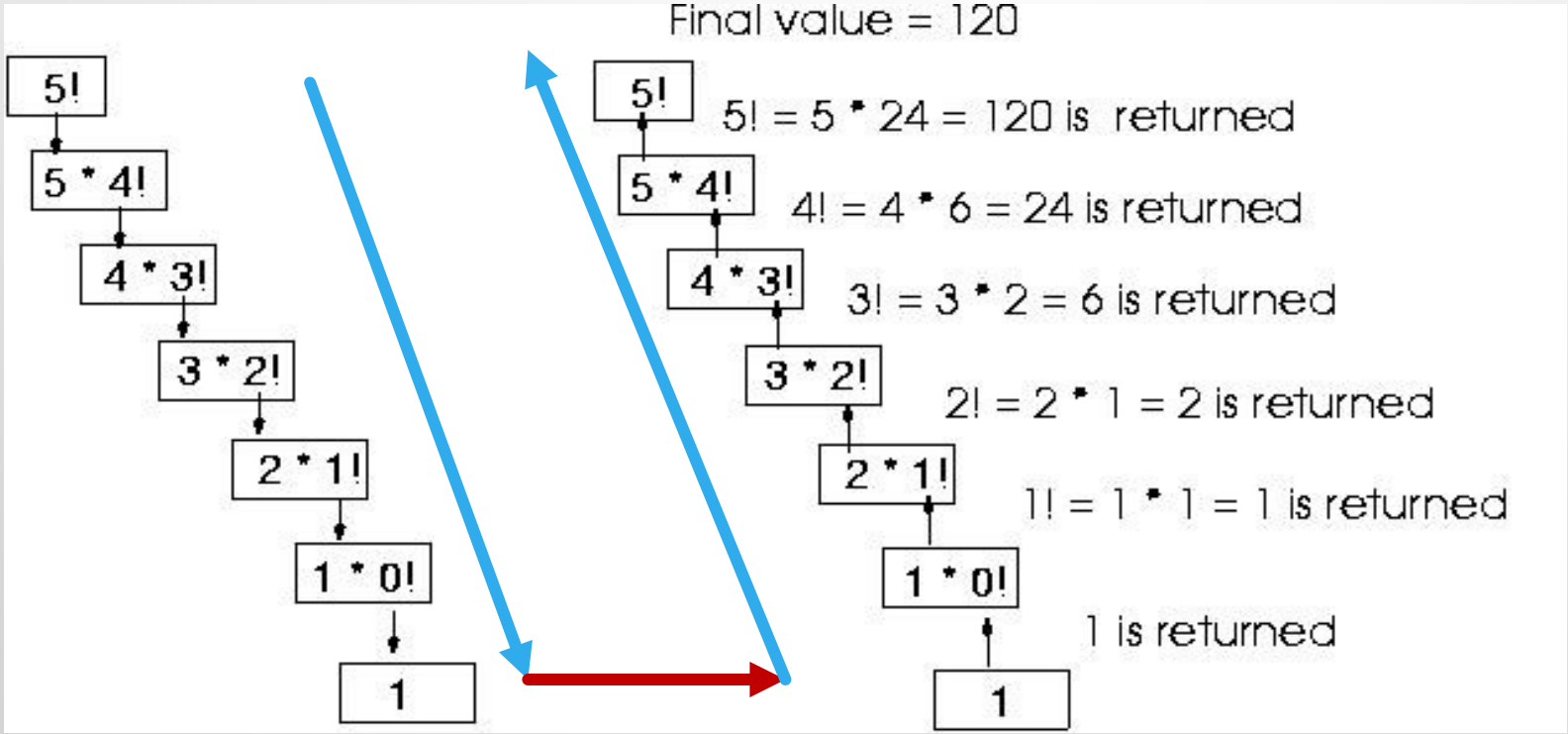
Hence, factorial can be solved iteratively:

```
# factorial, non-recursive (iterative)
def factorial(num):
    product=1
    for i in range(1,num+1):
        product*=i
    return product
x=eval(input('Enter a nonnegative number'))
print(factorial(x))
```

Recursive Programming- **Factorial**

Factorial can also be defined recursively. Let's see how it works recursively.

The below figure shows how **5!** is calculated recursively including all the recursive steps.



Recursive Programming- **Factorial**

Now, let's replace the notation *n*! with factorial(*n*), which looks a bit more like computer code, and rewrite these rules as follows:

If $n = 0$ then factorial(n) = 1
If $n > 0$ then factorial(n) = n * factorial($n - 1$)

```
def factorial(num):
    if num == 0:
        return 1
    else:
        return num * factorial(num-1)
x=eval(input('Enter a nonnegative number'))
print(factorial(x))
```

Base case

Recursive part

Consider what happens when the definition is applied to the problem of calculating **factorial (4)**:

factorial (4)= 4***factorial (3)**
= 4 *3* **factorial (2)**
= 4*3*2* **factorial (1)**
= 4*3*2* 1* **factorial (0)**
= 4*3*2*1*1 = **24**

Recursive Programming- **Exponents**

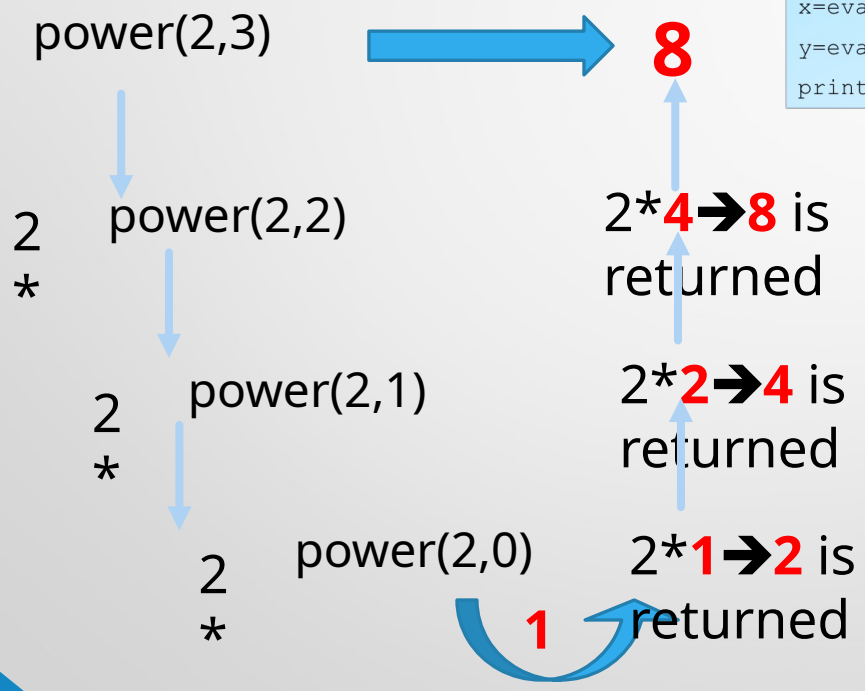
Let us consider the case of raising a real number, ***a***, to a nonnegative integer power, ***b***. ➔ a^b . A recursive method could be as follows:

```
#A program to calculate the value of 'a' to the power 'b'.
def power(a,b):
    if (b == 0):
        return 1
    else:
        return a * power(a, b-1)
x=eval(input('Enter a real number for the base '))
y=eval(input('Enter a nonnegative number for the exponent '))
print(power(x,y))
```

Recursive Programming- **Exponents**

Now , suppose that we want to calculate recursively ***power(2,3)*** using the function.

```
#A program to calculate the value of 'a' to the power 'b'.
def power(a,b):
    if (b == 0):
        return 1
    else:
        return a * power(a, b-1)
x=eval(input('Enter a real number for the base '))
y=eval(input('Enter a nonnegative number for the exponent '))
print(power(x,y))
```



Using Recursion

- Since each call to the recursive function reduces the problem:
  - Eventually, it will get to the base case which does not require recursion, and the recursion will stop
- Usually, the problem is reduced by making one or more parameters smaller at each function call

Direct and Indirect Recursion

The examples we have discussed so far show recursive functions or functions that directly call themselves.

- Direct recursion: when a function directly calls itself
  All the examples shown so far were of direct recursion

There is also the possibility of creating indirect recursion in a program.

- Indirect recursion: when function A calls function B which in turn calls function A.

The Fibonacci Series

Some mathematical problems are designed to be solved recursively.
One well-known example is the calculation of Fibonacci numbers.
The Fibonacci numbers, named after the Italian mathematician Leonardo Fibonacci (born circa 1170), are the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, . . .

Notice after the second number, each number in the series is the sum of the two previous numbers.
The Fibonacci series can be defined as follows:

- Fibonacci series: has two base cases and a recursive case:
  - `if n = 0 then Fib(n) = 0`
  - `if n = 1 then Fib(n) = 1`
  - **`if n > 1 then Fib(n) = Fib(n-1) + Fib(n-2)`**

The Fibonacci Series

A recursive function to calculate the *n*th number in the Fibonacci series is shown below:

```
# The fib function returns the nth number
# in the Fibonacci series
def fib(n):
     if n == 0:
       return 0
     elif n == 1:
         return 1
     else:
        return fib(n - 1) + fib(n – 2)
```

Notice this function actually has two base cases: when *n* is equal to 0, and when *n* is equal to 1.
In either case, the function returns a value without making a recursive call.

Finding the Greatest Common Divisor

- Calculation of the greatest common divisor (GCD) of two positive integers

  If x can be evenly divided by y, then

  gcd(x,y) = y

  Otherwise, **gcd(x,y) = gcd(y, remainder of x/y)**

- The corresponding function code will be:

```
# The gcd function returns the greatest common
# divisor of two numbers.
def gcd(x,y):
      if x % y == 0:
          return y
       else:
             return gcd(y, x % y)
```

Guidelines for Writing Recursive Methods

- A recursive method must have a well-defined termination or ***stopping state (****base case****)***.
- For the factorial method, this was expressed in the lines:

```
if (n == 0)
   return 1;
```

- The ***recursive step***, in which the method calls itself, must eventually lead to the stopping state.
- For the factorial method, the recursive step was expressed in the lines:

```
else
   return n * factorial(n - 1);
```

Guidelines for Writing Recursive Methods

- Because each invocation of the factorial method is passed a smaller value, eventually the stopping state must be reached.
- Had we accidentally written:

```
else
   return n * factorial(n + 1);
```

- The method would describe an **indefinite recursion**.
- Eventually, the user would notice and terminate the program, or else the Java interpreter would run out memory, and the program would terminate with a **stack overflow error**.

Recursion versus Looping

- Reasons not to use recursion:
  - Less efficient: entails function calling overhead that is not necessary with a loop. For example, we saw that n! can be calculated with a loop.
  - Usually, a solution using a loop is more evident than a recursive solution
- Some problems are more easily solved with recursion than with a loop
  - Example: Fibonacci, where the mathematical definition lends itself to recursion
- You must be able to determine when recursion is appropriate

Extra Exercises

Exercise 1

1. Design a function that accepts a list of numbers as an argument. The function should recursively calculate the sum of all the numbers in the list and return that value.

```
# The sum_list function accepts a list of numbers as an
argument.
# The function recursively calculates the sum of all the
# numbers in the list and returns the value.
def sum_list(numlist):
    n  = len(numlist)
    if len(numlist) == 1:
        return numlist[0]
    else:
        return numlist[n-1] + sum_list(numlist[0:n-1])
```

*Try to write the complete program that prompts the user to fill a list with **n** number of numbers, and use the above function to recursively calculate their sum.*

Exercise 2

2. Design a function that accepts a string as an argument. Assume that the string will contain a single word. The function should use recursion to determine whether the word is a palindrome (a word that reads the same backwards as forward).

Hint: Use string slicing to refer to and compare the characters on either

```
# Function uses recursion to determine whether
# the word passed to it is a palindrome.
def is_palindrome(word):
    if len(word) < 2:
        return True
    elif word[0] == word[-1]:
        return is_palindrome(word[1:-1])
    else:
        return False
```

Solve this exercise using iteration!

Summary

- This lecture covered:
  - Definition of recursion
  - The importance of the base case
  - The recursive case as reducing the problem size
  - Direct and indirect recursion
  - Examples of recursive algorithms
  - Recursion versus looping