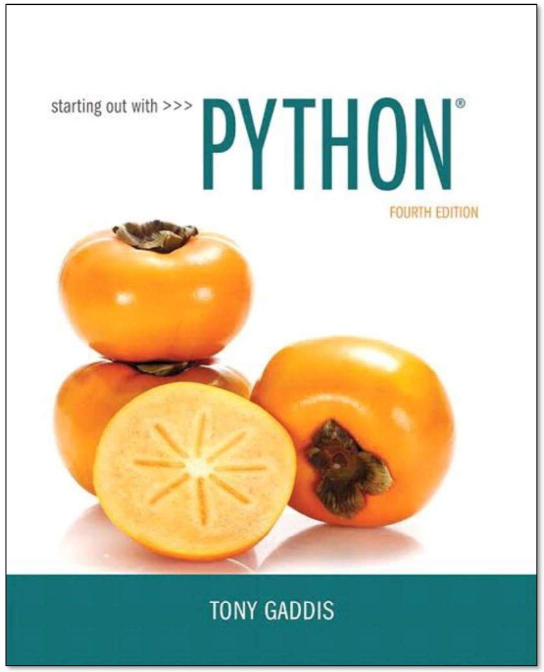
CHAPTER 12

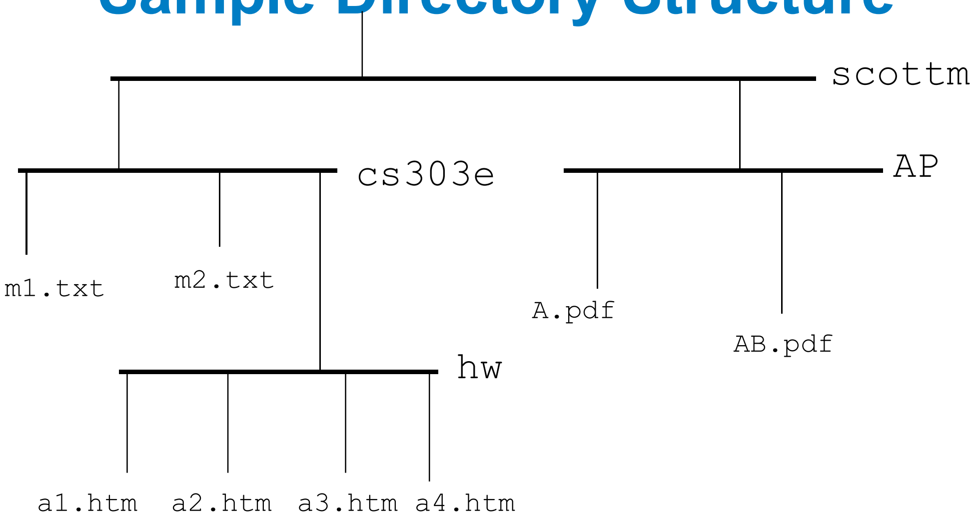
Recursion



An Interesting Problem

- **Write a method that determines how much space is take up by the files in a directory**
- **A directory can contain files and directories**
- **How many directories does our code have to examine?**
- **How would you add up the space taken up by the files in a single directory**
  - Hint: don't worry about any sub directories at first

Sample Directory Structure



os.path

- **We used os.path to check if a path (location of a file or directory) refers to a file that exists**
- **Lots of other useful methods:**
  - os.path.isfile(path)
  - os.path.isdir(path)
  - os.path.getsize(path)
    - Return the size, in bytes, of path. Raise OSError if the file does not exist or is inaccessible.
  - os.listdir(path='.')
    - Return a list containing the names of the entries in the directory given by path.

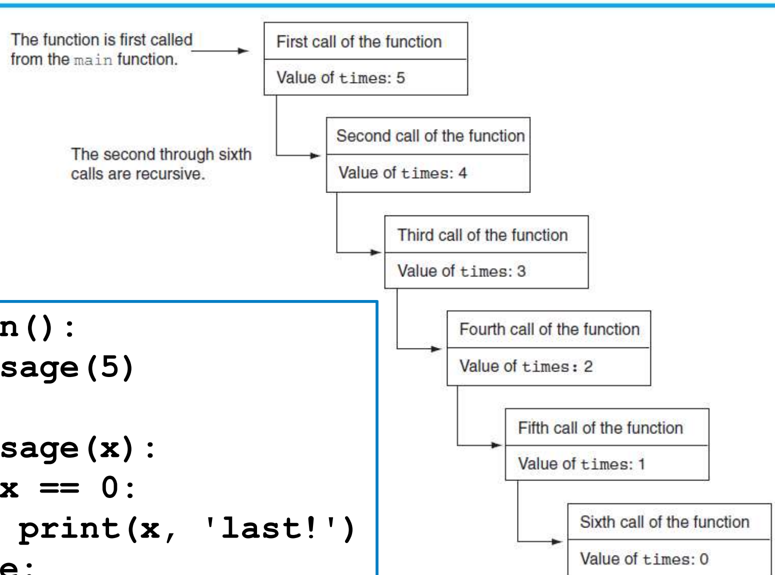
Implementation

- **Write a function that given the name of a directory returns the size of the files in that directory**
  - … and if the directory has directories in it (subdirectories) return the size of the files in those subdirectories
    - … and if those subdirectories have subdirectories…

Introduction to Recursion

- **<u>Recursive function</u>: a function that calls itself (with different arguments)**
- **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
- **<u>Depth of recursion</u>: the number of times a function calls itself**

Figure 12-2 Six calls to the message function

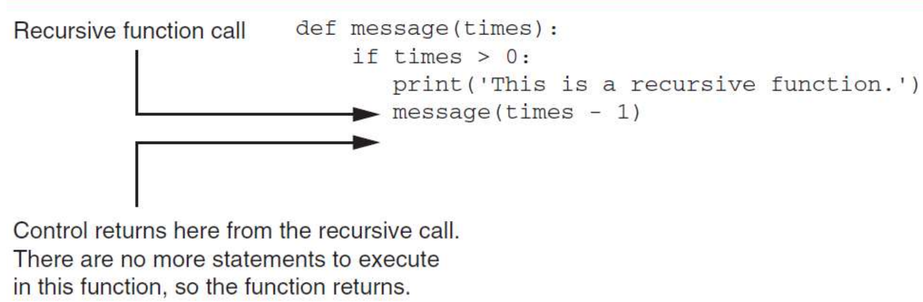



```
def main():
    message(5)

def message(x):
    if x == 0:
        print(x, 'last!')
    else:
        print(x)
        message(x - 1)
```


Introduction to Recursion (cont'd.)

Control returns to the point after the recursive function call



Problem Solving with Recursion

- **Recursion is a powerful tool for solving repetitive problems**
- **Recursion is never *required* to solve a problem**
  - Any problem that can be solved recursively can be solved with a loop
  - Recursive algorithms may be less efficient than iterative ones in the number of computations
    - Due to *overhead* of each function call

Problem Solving with Recursion (cont'd.)

- **Some repetitive problems are more easily solved with recursion**
- **General outline of recursive function:**
  - If the problem can be solved now without recursion, solve and return
    - Known as the *base case*
  - Otherwise, reduce problem to smaller problem of the same structure and call the function again to solve the smaller problem
    - Known as the *recursive case*

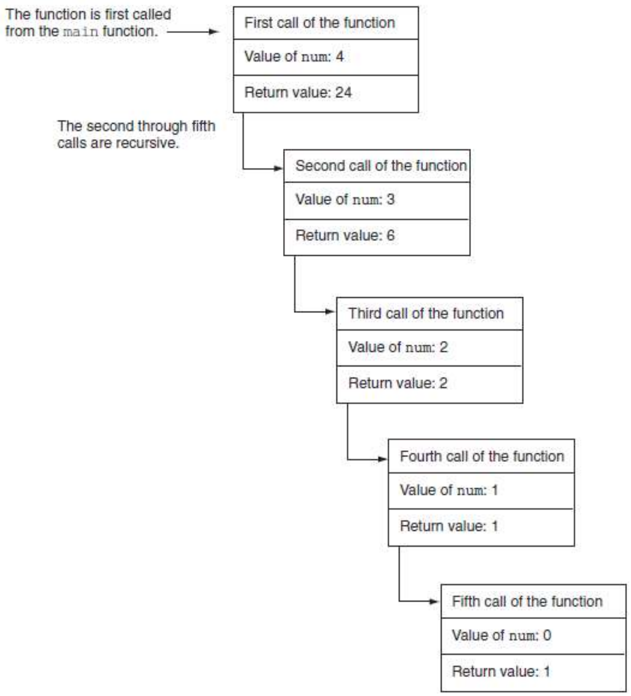
Using Recursion to Calculate the Factorial of a Number

- **In mathematics, the $n!$ notation represents the factorial of a number *n***
  - 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
    - factorial(n) = n x factorial(n-1)

Using Recursion (cont'd.)

```
# The factorial function uses recursion to
# calculate the factorial of its argument,
# which is assumed to be nonnegative.
def factorial(num):
    if num == 0:
        return 1
    else:
        return num * factorial(num - 1)
```

Figure 12-4 The value of num and the return value during each call of the function



Using Recursion (cont'd.)

- **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

- **Direct recursion: when a function directly calls itself**
  - All the examples shown so far were of direct recursion
- **Indirect recursion: when function A calls function B, which in turn calls function A**
  - also known as mutual recursion

Examples of Recursive Algorithms

- **Summing a range of list elements with recursion**
  - Function receives a list containing range of elements to be summed, index of starting item in the range, and index of ending item in the range
  - Base case:
    - `if start index > end index return 0`
  - Recursive case:
    - `return current_number + sum(list, start+1, end)`

Examples of Recursive Algorithms (cont'd.)

```
# The range_sum function returns the sum of a specified
# range of items in num_list. The start parameter
# specifies the index of the starting item. The end
# parameter specifies the index of the ending item.
def range_sum(num_list, start, end):
    if start > end:
        return 0
    else:
        return num_list[start] + range_sum(num_list, start + 1, end)
```


The Fibonacci Series

- **Fibonacci series: has two base cases**
  - `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)`
- **Corresponding function code:**

```
# 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)
```


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)
- **Corresponding function code:**

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


The Towers of Hanoi

- **Mathematical game commonly used to illustrate the power of recursion**
  - Uses three pegs and a set of discs in decreasing sizes
  - Goal of the game: move the discs from leftmost peg to rightmost peg
    - Only one disc can be moved at a time
    - A disc cannot be placed on top of a smaller disc
    - All discs must be on a peg except while being moved

The Towers of Hanoi (cont'd.)

Figure 12-5 The pegs and discs in the Tower of Hanoi game

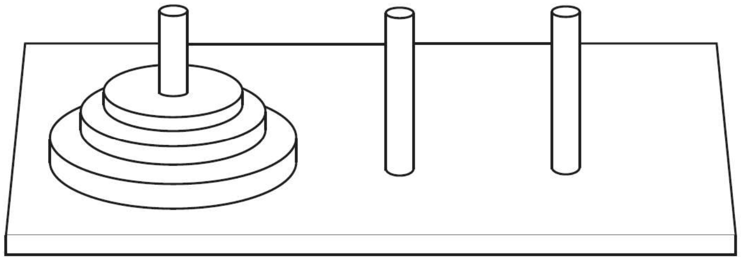
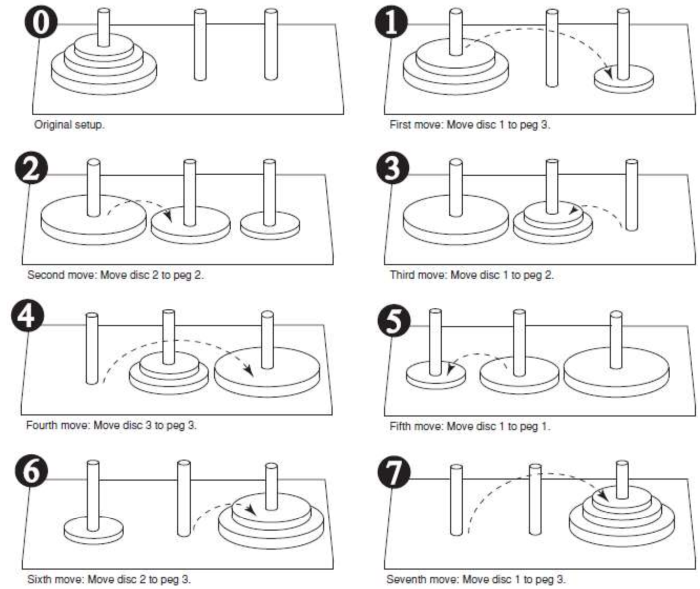


Figure 12-6 Steps for moving three pegs



The Towers of Hanoi (cont'd)

- **Problem statement**: move n discs from peg 1 to peg 3 using peg 2 as a temporary peg
- **Recursive solution**:
  - If n == 1: Move disc from peg 1 to peg 3
  - Otherwise:
    - Move n-1 discs from peg 1 to peg 2, using peg 3
    - Move remaining disc from peg 1 to peg 3
    - Move n-1 discs from peg 2 to peg 3, using peg 1

The Towers of Hanoi (cont'd.)

```
# The moveDiscs function displays a disc move in
# the Towers of Hanoi game.
# The parameters are:
#     num:       The number of discs to move.
#     from_peg:  The peg to move from.
#     to_peg:    The peg to move to.
#     temp_peg:  The temporary peg.
def move_discs(num, from_peg, to_peg, temp_peg):
    if num > 0:
        move_discs(num - 1, from_peg, temp_peg, to_peg)
        print('Move a disc from peg', from_peg, 'to peg', to_peg)
        move_discs(num - 1, temp_peg, to_peg, from_peg)
```

Recursion versus Looping

- **Reasons not to use recursion:**
  - Less efficient: entails function calling overhead that is not necessary 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: Factorial, where the mathematical definition lends itself to recursion