Recursive Functions

So far we have used functions that called other functions, but never did call themselves.

A Recursive function is a function that calls itself. But if it calls itself, then when does it stop calling itself and start to return (i.e., it has to return through the whole sequence of calls it makes to itself.

To step calling itself forever, a recursive function needs a statement called a BASE CASE. The BASE CASE tells it whether to stop (and start the chain of returns) or continue with another call to itself.

A function that uses a loop to do some computation (i.e., it does not call itself) is an ITERATIVE function (i.e., it is non-recursive).

Let's take a look at an iterative function — the kind we already know.

Example: We will write an iterative function to print all the integers starting at integer low and stopping at integer high.

```python
def iRange(low, high): # iterative function to print range of integers
    ''' iteratively print all integers, from low to high.''
    while (low <= high): # the loop will stop as soon as j increases past high
        print(low)
        low = low + 1
```

Note these two points: 1. local variable “low” keeps getting incremented; local variable “high” does not change at all 2. loop continues as long as low <= high

Now let’s try to do this by making the function call itself. Remember that it must have a BASE CASE so that it can stop, and start the sequence of returns.

```python
def rRange(low, high): # recursive function to print range of integers
    ''' print all integers, from low to high.''
    if (low <= high): # the loop condition from iRange becomes our base case; “low” keeps
```
# getting incremented in each recursive call; finally this test fails

print(low)

# and the recursive calling stops

rRange(low+1, high) # notice how we enter the recursive call with a “smaller” problem

# Note: Both iRange and rRange go through the exact same sequence of calls and produce # the same values s output

def main():
    
    iRange(11,22);
    print("\n\n")
    rRange(11,22)

    main()

# 2.py

# In the previous example we only printed things recursively. Now let's try to recursively # create values and return them.

# We used this example earlier, when we first defined functions.

# Problem: write a function sum(low, high) that computes and returns the sum of integers # low + (low + 1) + (low + 2) + ....... + high; it returns 0 if called with low > high

# Recursion step: low + sum(low + 1, high)

# Observe how the recursion step replaces the orginal problem sum(low, high) with the subproblem # low + sum(low + 1, high). This is the key idea in recursion, but do not forget the base step.

def sum(low, high):
    ''' Returns the sum of all integers from low to high.'''

    if (low > high): # notice how this base case test gives us a way to stop recursing
        return 0

    else:
        return ( low + sum (low + 1, high))   # farming off a subproblem to next call

' Using recursion involves the “divide-and-conquer” strategy. You are given a problem in terms of n Try to reduce that problem to a combination of a partial solution and the same problem in terms of a smaller n (typically n-1,
or perhaps n2). In the above example we added “low” to the result of a smaller problem with one less number in the sum. Instead of going from low to high, the smaller subproblem went from (low + 1) to high. ’ # # TRACING a recursive function #3.py # Let's trace (i.e., see how the calls are sequenced and how the returns are sequenced) a # recursive function call # To help see the pattern we must do some pretty-printing # Every time we make a recursive call we will print the arguments to the call. But also, every time # we go deeper into the recursion, we will move the printing to the right, so the pattern is clear. # When the base case is hit and the recursive function starts to return, we will do the same # printing but in reverse, but instead of printing the arguments, we will print the result that # each particular call returns …. until we return to the first call which gives the final result. def sum (low, high, margin): #add all integers between low and high ' Returns a sum of the numbers from low to high. Print the recursive call parameters and the return results in such a way that the recursive pattern is clear.' blanks = “” * margin # margin tells us how many spaces to the right or left we must print print(blanks, low, high) # print margin spaces, then print arguments to first call if (low > high): # we have reached the bottom of the recursive calls if this is true print (blanks, 0) # sum cannot compute anything if low > high. So print 0. return 0 # since nothing to compute, there is nothing to return. Return 0. else: result = low + sum (low + 1, high, margin + 4) # please study this recursion # low + new subproblem # printing moves 4 spaces to the right print(blanks, result) return (result) # Hint: tracing in this way can help you debug recursion because the pattern will not # confuse you. Make sure you understand that the prints going to the right make up the # sequence of recursive calls. The prints going to the left mak up the returns up this chain # of calls. def main(): sum(1, 7, 0) # Mapping a math recursion to a recursive Python function #4.py # The famous Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, ….. # F(1) = 1, F(2) = 1, F(3) = 2, F(4) = 3, F(5) = 5, F(6) = 8, F(7) = 13, ….. # The first two numbers are 1, 1 # from then on, the next number is the sum of the previous two. # Let's write an iterative version of this function first def iFib(n): # returns the nth Fib number in the sequence if (n == 1): return 1 elif (n == 2): return 1 else: previous = 1 # save the previous and the current = 1 # current fib numbers index = 3 while (index ⇐ n): next = previous + current # now start to add the previous two to get the current previous = current current = next # update the current and previous index = index + 1 return next ' Now let's look at the math (recursive) definition of Fibonacci F(n) = 1 , when n = 1 or n = 2 F(n) = F(n-1) + F(n-2) when n > 2 and observe how it easily maps into recursive Python code.''' def rFib(n): # recursive definition of Fibonacci, using the math recursion if ( n < 3): return 1 else: return (rFib(n-1) + rFib(n-2)) def main(): print("Iteration \n") for j in range (1,12): print(iFib(j)) print("\n\n") print("Recursion \n") for j in range (1,12): print(rFib(j)) main() # recursive factorials #5.py # We know that n! = n(n-1)(n-2)(n-3)……. 3.2.1. Let's compute it recursively. def rFact(n): # returns n! if (n ==1): return 1 else: return(n*rFact(n-1)) # Home-work: write the iterative version to check results # n! gives you the number of ways of permuting n objects. # What if you wanted to (do combinations) select objects. How many ways are there of # selecting k objects from n objects? This is nCk or (n,k) …. and …. # the answer is : n! / (k! (n-k)!) # so you see that to compute this, you need 3 factorials. def Comb(n,k): # number of
ways of choosing $k$ objects from $n$ objects

$$\text{denom} = rFact(k) \times rFact(n-k) \quad \text{num} = rFact(n)$$

#1

num
denom)) # # Now supposing you had a string of $n$ arbitrary objects .... let's say 'abcdefg'. How
# can we list out all arrangements of $k$ objects at a time? def combinations(string, $k$): # want list of all
# $k$ things from string
outlist = [] # start with empty list and build it recursively for $j$ in range
(len(string)): if ($k == 1$): outlist.append(string[$j$]) # if there's only one object, put it in the list # Next
slice through the smaller sublist recursively, and concatenate the # new elemen to each sub
arrangement for element in combinations(string[$j+1$: ], $k-1$): outlist.append(string[$j$] + element)
return outlist

#HW: Also read the example on anagrams (page 437) .... it is similar. def main():
print("Factorial 5 = ", rFact(5)) print("Factorial 10 = ", rFact(10)) print("Factorial 27 = ", rFact(27))
print("Number of ways of choosing two objects from ten objects = ", Comb(10,2))
print("Combinations:", combinations("ABCDEFGH",3))

main() 

Recursive Binary Search 

When you trace the possible paths a binary search can take, you get a tree. # It starts with a single
node with $n$ elements. After one step you are down by one level, # with two nodes, each having
roughly $n/2$ elements. As in the first step, when you go down # further by one level, each node again
splits into two nodes with roughly $n/4$ items each. # In this way, in $\log(n)$ steps (log to the base 2) we
arrive at $n$ nodes, each containing # a single list element, and by this step the search problem has
been solved. # Whenever you see a problem that exhibits this tree-pattern you know you have a #
candidate for recursion. def rBinSearch(x, nlist, low, high): # search for $x$ from $nlist[low]$ to $nlist[high]$ if (low > high): # no more work return -1

mid = (low + high) / 2

item = nlist[mid] # is this $x$? or is it
smaller? or is it larger?

if (item == x):
return mid # found it!

elif (x < item):
# if $x$ is in $nlist$, it's
in the left sublist

return rBinSearch(x, nlist, low, mid -1)

else:
# or since its greater
than item, it's in the right sublist

return rBinSearch(x, nlist, mid + 1, high)

def main():

nlist = [1, 5, 7, 19, 21, 27, 31, 45, 56, 63, 66, 71, 73, 79, 84, 85, 89, 93, 97, 99]

index = rBinSearch(7, nlist, 0, 19)

if (index > 0):
print("Found the number in location ", index)

main() 

Computing things with exponents very quickly 

In your lab you will do this with matrices. The idea is the same. Here we will use numbers. In # lab you replace the numeres by
matrices. No conceptual difference when taking powers. # Idea: you want to compute x raised to the power 10. # One way is to multiply it out 10 times like this def power(x, n): # x raised to the power n

```python
result = 1
for i in range(n):
    result = result * x  # if the power is n, you do n multiplications in this approach
return(result)
```

# Faster way: x^5 * x^5 = x^10, so we can get x^10 with one multiplication if we had x^5 # x^4 = x^2 * x^2, so things are looking good # first get x^2 which is x*x # then get x^4 which is x^2*x^2 # then get x^5 which is x^4*x (so far we have done 3 multiplications in total) # finally get x^10 which is x^5*x^5 —> presto! in 4 multiplications we have the answer # This is a lot faster than doing 10 multiplications to get x^10 # The only (small) problem is that the power can be odd or even. See the simple formula # at the bottom of page 438 in your text. def rpower(x, n): # c raised to the power n, recursively, fast

```python
if ( n == 0):
    return 1
else:
    factor = rpower(x, n//2)  # get half the power recursively
    if (n % 2 == 0):
        return (factor*factor)  # even number, no extra multiplication needed
    else:
        return(factor*factor*x)  # need an extra multiplication because n is odd
```

# Iterative and Recursive string reversal #8.py # Let's write a simple iterative function that takes in a string, reverses it and returns # the reverse string. It uses a new string to do this. def iReverse_with_new(s): # s is the input string

```python
new = ""
j = len(s) - 1  # this is the position of last character in string
while (j >= 0):
    new = new + s[j]  # built the new string starting from last character, backwards
    j = j - 1
return new
```

# Now let's try it recursively def rReverse(s):
```python
# Idea: if there is only one character in the string, return it; there is no work to do.

# suppose the length of the string is n
# otherwise, put the first character in the n-th position, and recursively call the function to work on the first (n-1) characters

if (len(s) <= 1): # if string has only one element or is empty
    return s
else:
    return (rReverse(s[1:]) + s[0]) # put first char in last position and call rReverse on the rest of the string, recursively

def main():

    print("iReverse_with_new")
    print(" ")
    print(iReverse_with_new("This is a small string"))
    print(iReverse_with_new("0123456789"))
    print(iReverse_with_new(" "))
    print(iReverse_with_new("0 1"))
    print(iReverse_with_new(""))

    print("\nrReverse")
    print(" ")
    print(rReverse("This is a small string"))
    print(rReverse("0123456789"))
    print(rReverse(" "))
    print(rReverse("0 1"))
    print(rReverse(""))

main() </code>

From:  
http://courses.cs.purdue.edu/ - Computer Science Courses

Permanent link:  

Last update: 2016/04/14 14:53