Python Intermediate Programming

Unit 2: Basic Python Algorithms

CHAPTER 4: RECURSIVE FUNCTIONS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- In this lecture we will discuss the concept of recursion and examine recursive functions that operate on integers, strings, and lists, learning common idioms for each. As with other topics discussed this quarter that you have already seen, I want to ensure that you have a deep understanding of recursion.

Objectives

- The concept of recursively defined (sometimes called inductively defined) data types and recursion is fundamental in many areas of computer science, and this concept should be discussed from many angles in many of your ICS classes; you should become comfortable with seeing and applying recursion. In addition, some programming languages (Lisp and Haskell are the foremost examples) use recursion (and also decision: if) as their primary control structures: any iterative code can be written recursively (and recursion is even more powerful than iteration, as we glimpsed in the EBNF lecture).
- Even languages that are not primarily recursive all support recursion (and have since the late 1960s), because sometimes using recursion is the best way to write code to solve a problem: best often means simplest, but sometimes it can mean most efficient too (efficient in writing and/or efficient in running).

Objectives

- Python (and C/C++/Java) are not primarily recursive languages. Each has strong features for iterating through data (Python has the most powerful tools for such iteration, including generators).
- But, it is important that we learn how to write recursive code in Python too. Next week, we will recursively define the linked list and binary tree data structures and see how to manipulate them, both iteratively (for some functions) and recursively (for all functions). In ICS-46 we will revisit these data structures (and more) many times using C++, and again see how we can manipulate them both iteratively and recursively.

Objectives

- Douglas Hofstadter's Pulitzer-prize winning book, "Godel, Escher, Bach" is an investigation of cognition, and commonly uses recursion and self-reference to illustrate the concepts it is discussing. It is a fascinating book and because it is old, can be purchased quite cheaply as a used book.

http://www.amazon.com/G%C3%B6del-Escher-Bach-Eternal-Golden/dp/0465026567

(at least read some small reviews of this book here). I recommend it highly.

Recursion vs Iteration

LECTURE 1

Recursion vs Iteration

- Recursion is a programming technique in which a call to a function results in another call to that same function. In direct recursion, a call to a function appears in the function's body; in indirect/mutual recursion, the pattern is some function calls some other function ... which ultimately calls the first function. In an example where `f` calls `g` and `g` calls `f`, we say `f` and `g` are mutually recursive with `f` calling `f` indirectly via `g`, and `g` calling `g` indirectly via `f`.

Recursion vs Iteration

- For some data structures (not many built-into Python) and problems, it is simpler to write recursive code than its iterative equivalent. In modern programming languages, recursive functions may run a bit slower (maybe 5%) than equivalent iterative functions, but this is not always the case (and sometimes there is no natural/simple iterative solution to a problem); in a typical application, this time difference is insignificant (most of the time will could be spent elsewhere anyway).

Recursion vs Iteration

- We will begin by studying the form of general recursive functions; then apply this form to functions operating on int values, and then apply this form to functions operating on strings and lists. In all these cases, we will discuss how values of these types are recursively defined and discuss the natural "sizes" of the problem solved.
- To start, suppose that we have the problem of collecting $1,000.00 for charity, with the assumption that when asked, everyone is willing to chip in the smallest amount of money: a penny.

Recursion vs Iteration

Iterative solution : visit 100,000 people, and ask each for a penny

Recursive solution:

if you are asked for a penny, give a penny to this person

otherwise

visit 10 people and ask them each to collect 1/10 the amount that you are asked to raise; collect the money they give you into one bag; give this bag to the person who asked you

- In the iterative version each subproblem is the same; raising a penny. In the recursive solution, subproblems get smaller and smaller until they reach the problem of collecting a penny (they cannot get any smaller: this problem has the smallest size because there is no smaller currency).

Recursion vs Iteration

The general form of a directly recursive function is

def Solve(Problem):

if (Problem is minimal/not decomposable into a smaller problem: a base case)

Solve Problem directly and return solution; i.e., without recursion

else:

1. Decompose Problem into one or more SIMILAR, STRICTLY SMALLER subproblems: SP1, SP2, ... , SPn
2. Recursively call Solve (this function) on each smaller subproblem (since they are similar): Solve(SP1), Solve(SP2),..., Solve(SPN)
3. Combine the returned solutions to these smaller subproblems into a solution that solves the original, larger Problem (the one this function call must solve)
4. Return the solution to the original Problem

Simple Recursion in Python

LECTURE 2

Simple Recursion in Python

- We will start by examining a recursive definition for the factorial function (e.g., 5! reads as "five factorial") and then a recursive function that implements it. The definition is recursive because we define how to compute a big factorial in terms of computing a smaller factorial. Note that the domain of the factorial function is the non-negative integers (also called the natural numbers), so 0 is the smallest number on which we can compute factorial.

  0! = 1

  N! = N*(N-1)! for all N>0; recursive: we define ! in terms of a smaller !

- By this definition (and just substitution of equals for equals) we see that

  5! = 5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 5*4*3*2*1*0! = 5*4*3*2*1*1

- We have eliminated all occurrences of !, so we can use just * to compute

  5! = 5*4*3*2*1*1 = 120.

Simple Recursion in Python

- The first definition below is a transliteration of the general code above, decomposing it into just one similar (factorial) but simpler (n-1) subproblem.

```
def factorial (n):
    if n == 0:
        return 1
    else:
        sub_problem         = n-1
        solved_sub_problem = factorial(sub_problem)
        solved_original_n  = n*solved_sub_problem
        return solved_original_n
```

Simple Recursion in Python

- The next definition is a simplification of how this function should really be written in Python, without all the intermediate names, which are not needed and don't really add any clarity.

```
def factorial (n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Simple Recursion in Python

- This definition looks clean and closely mirrors the recursive mathematical description of factorial. In fact, because of the simplicity of this particular recursive function, we can write an even simpler solution using a conditional expression; but I prefer the solution above, because it is more representative of other recursive solutions (to more complicated problems).

```
def factorial(n):
    return (1 if n == 0 else n*factorial(n-1))
```

Simple Recursion in Python

- We can contrast the recursive code with the iterative code that implements the factorial function

```
from goody import irange
def factorial (n):
    answer = 1;
    for i in irange(2,n)
        answer *= i
    return answer
```

Simple Recursion in Python

- Note that this function defines two local names (answer and i) and binds 1 to answer and rebinds it to a new value during each execution of the for loop's body. Likewise, i is rebound to a sequence of values produced when the function iterates over the `irange(2,n)`. The recursive function defines no local names and doesn't rebind any names (although each recursive call binds an argument to the parameter in the new recursive function call).

Simple Recursion in Python

- Rebinding the values of names make it hard for us to think about the meaning of code (they make it tougher to prove that the code is correct too), and makes it hard for multi-core processors to coordinate in solving a problem. "Functional programming languages" (those that allow binding of a name to computed value, but no rebinding to that names) are more amenable to be automatically parallelizable (can run more quickly on multi-core computers). You'll see more about this in later classes at UCI (e.g., Concepts of Programming Languages).

Simple Recursion in Python

- We can mimic factorial's recursive definition for a function that raises a number to an integer power. Note that the domain of n for this power function requires n to be a natural number (a non-negative integer).

```
A**0 = 1 (yes, this is even true when A=0)
A**N = A * A**(N-1)  for all N>0
```

So,

```
A**4 = A * A**3 = A*A * A**2 = A*A*A * A**1
     = A*A*A*A * A**0 = A*A*A*A*1
```

Simple Recursion in Python

- We can likewise translate this definition into a simple recursive Python function

```
def power(a,n):
    if n == 0:
        return 1
    else:
        return a*power(a,n-1)
```

- By this definition (and just substitution of equals for equals) we see that calling power(a,n) requires n multiplications.

```
power(a,3) = a*power(a,2) = a*a*power(a,1)
           = a*a*a*power(a,0) = a*a*a*1
```

Simple Recursion in Python

- Of course, we could write this code iteratively as follows, which also requires n multiplications

```
def power(a,n):
    answer = 1
    for i in irange(1,n):
        answer *= a
    return answer
```

- But there is another way to compute power(a,n) recursively, shown below. This longer function requires between Log2 n and 2*Log2 n multiplications. Here Log2 means the log function using a base of 2. Note Log2 1000 is about 10 (2**10 = 1,024), so Log2 1,000,000 is about 20, Log2 1,000,000,000 is about 30): so, to compute power(a,1000) requires between 10 and 20 multiplications (not the 1,000 multiplications required by the earlier definitions of power).

Simple Recursion in Python

```
def power(a,n):
    if n == 0:
        return 1
    else:
        if n%2 == 1:
            return a*power(a,n-1)
        else:
            temp = power(a,n//2)
            return temp*temp
```

Simple Recursion in Python

- Here we bind temp ONCE(and never rebind it) and then use its value, which is fine for functional programming. We could get rid of the local name temp completely by defining the local function def square(n): n*n inside power and then calling it in the else clause: return square(power(a,n//2)), or we could just the "raise to a power" operator: **

```
def power(a,n):
    def square(n): return n*n
    if n == 0:
        return 1
    else:
       if n%2 == 1:
           return a*power(a,n-1)
       else:
           return square( power(a,n//2) )
```

Simple Recursion in Python

For one example

- power(a,16) computes power(a,8) and returns its result with 1 more multiplication; power(a,8) computes power(a,4) and returns its result with 1 more multiplication; power(a,4) computes power(a,2) and returns its result with 1 more multiplication; power(a,2) computes power(a,1) and returns its result with 1 more multiplication; power(a,1) computes a*power(a,0), which requires 1 multiplication: computing power(a,0) requires 0 multiplications (it just returns a value).
- In all, power(a,16) requires just 5 multiplications, not 16. Note that this function is NOT guaranteed to always use the minimum number of multiplications. Power(a,15) uses 6 multiplication, but by computing x3 = x*x*x then x3*(square(square(x3))) requires only 5: see the topic named "addition-chain exponentiation" if you are interested in what is known about the minimal number of multiplications required for exponentiation. No simple algorithms solves this problem in the minimum number of multiplications.

Simple Recursion in Python

- We will prove that this function computes the correct answer later in this lecture. Truth be told, we can write a fast power function like this iteratively too, but it looks much more complicated and is much more complicated to analyze its behavior and prove that it is correct.

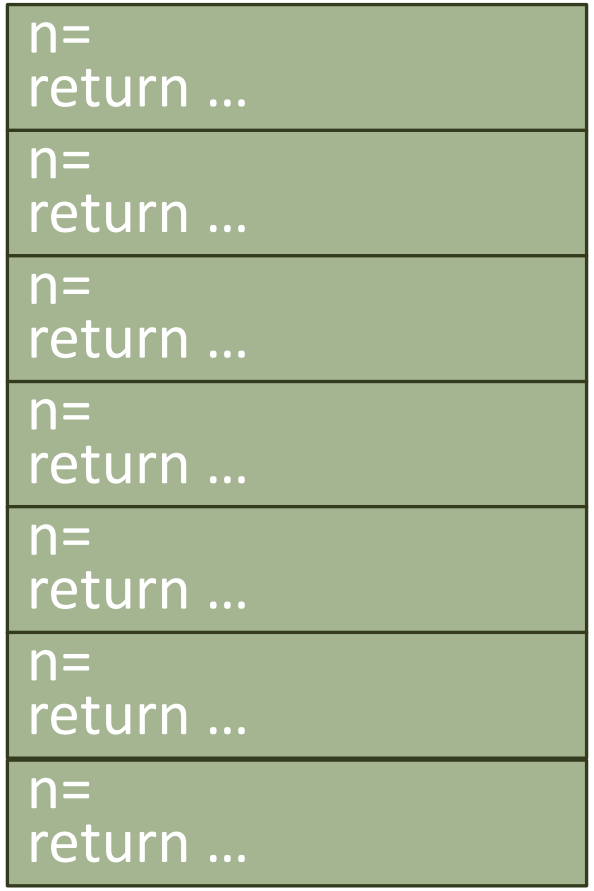
Hand Simulation

LECTURE 3

Hand Simulation

- Next, we will learn how to hand-simulate a recursive functions using a "tower of call frames" in which each resident in an apartment executes the same code (acting as the function) to compute a factorial: he/she is called by the resident above and calls the resident underneath, when a recursive call is needed (calling back the resident above when their answer is computed). While it is useful to be able to hand-simulate a recursive call, to better understand recursion, hand-simulation is not a good way to understand or debug recursive functions (the 3 proof rules discussed below are a better way). I will do this hand simulation on the document camera, using the following form for computing factorial(5).

Factorial Towers



Proof Rules for Recursive Functions

LECTURE 4

Proof Rules

- Now, we will learn how to verify that recursive functions are correct by three proof rules. Even more important than proving that existing functions are correct (to better understand them), we will use these same three proof rules to guide us when we synthesize new recursive functions.
- Note that in direct recursion, we say that the function "recurs", not that it "recurses". Recurses describes what happens when you hit your thumb with a hammer the second time. Programmers who use the words "recurse" or "recurses“ are not well-spoken.

Proof Rules

- The three proof rules should be simple to apply in most cases. These rules mirror rules for proofs by induction in mathematics.
    1. Prove that the base case (smallest) problem is processed correctly. Should be easy, because base cases are tiny and their solutions simple.
    2. Prove that each recursive call is on a smaller-sized problem: the problem gets closer to the base case. Should be easy because there are "standard" ways to recur: ints decrease by 1 or a factor of 10 (i.e., x//10 has one fewer digit; x%10 has one digit); Strings, tuples, and lists recur on a slices (fewer characters, fewer values).
    3. ASSUMING ALL RECURSIVE CALLS SOLVE THEIR SMALLER SUBPROBLEMS CORRECTLY, prove that the code combines these solved subproblems correctly, to solve the original Problem (the parameter of the function). Should be easy, because we get to assume something very important and powerful: all subproblems are solved correctly.

Proof Rules

Here is a proof, using these 3 rules, that the factorial function is correct:

1) The base case is 0; and according to the recursive mathematical definition, 0! = 1. This function recognizes an argument of 0 and returns the correct value 1 as the result.
2) If n is a non-negative number that is not 0 (not the base case), then this function makes one recursive call: n-1 is a smaller-sized problem, closer to 0 (the base case) than n is. It is closer by 1: the distance between n-1 and 0 is 1 less than the distance between n and 0.
3) ASSUMING factorial(n-1) COMPUTES (n-1)! CORRECTLY, this function returns n*factorial(n-1), which is n*(n-1)! by our assumption, which according to the mathematical definition is the correct answer for n!, the parameter to the function call.

Proof Rules

- Notice that the focus of the proof is on ONE call of the function (not like the hand simulation method above, which looked at all the recursive calls). We look at what happens if it is a base case (1)or if it actually recurs (2-3). For the recursive case, we don't worry about more recursive calls, because we get to assume that any further recursive calls (on smaller problems, which might be the base case or at least closer to the base cases) compute the correct answer WITHOUT HAVING TO THINK about what happens during any recursive calls.

Proof Rules

- Proof that fast-power function is correct (the code is duplicated from above):

```
def power(a,n):
    def square(n): n*n
    if n == 0:
        return 1
    else:
       if n%2 == 1:
           return a*power(a,n-1)
       else:
           return square( power(a,n//2) )
```

Proof Rules

1) The base case is 0; and according to the recursive mathematical definition, a**0 = 1. This function recognizes an argument of 0 and returns the correct value 1 for it.

2) If n is a non-negative number that is not 0 (not the base case), then if n is odd, n-1 is a smaller-sized problem: closer to 0 (the base case) than n is; if n is even (it must be >= 2), n//2 is also a smaller-sized problem: closer to 0 (the base case) than n is.

Proof Rules

3) ASSUMING power(a,n-1) COMPUTES a**(n-1) CORRECTLY AND power(a,n//2) COMPUTES a**(n//2) CORRECTLY. We know that any n must be either odd or even: if n is odd, this function returns a*a**(n-1), so it returns (by simplifying) a**n, which is the correct answer for the parameters to this function; likewise, if n is even, this function returns the value square(a**(n//2)), which returns (by simplifying) a**n, which is the correct answer for the parameters to this function. For all even numbers n, n//2 is half that value, with no truncation: for example, for n the even number 10, square (a**(10//2)) = square (a**5) = (a**5)**2 = a**10.

Proof Rules

- Again, the focus of the proof is on one call of the function: the parts concern only the base case and the recursive case (now two cases, depending on whether or not n is odd or even): and for the recursive cases, we don't worry about more recursive calls, because we get to assume that any recursive calls (on smaller problems, closer to the base cases) compute the correct answer without having to think about what happens during the recursion.
- What happens if we write factorial incorrectly? Will the proof rules fail. Yes, for any flawed definitions one will fail. Here are three examples (one failure for each proof rule).

Proof Rules

```
def factorial (n):
    if n == 0:
        return 0              # 0! is not 1
    else:
        return n*factorial(n-1)
```

Proof Rules

- This factorial function violates the first proof rule. It returns 0 for the base case; since everything is multiplied by the base case, ultimately this function always returns 0. Bar bet: you name the year and the baseball team, and I will tell you the product of all the final scores (last inning) for all the games they played that year. How do I do it and why don't I make this kind of bet on basketball teams?

```
def factorial (n):
    if n == 0:
        return 1
    else:
        return factorial(n+1)//(n+1)
        # n+1 not closer to base case: 0
```

Proof Rules

- This factorial function violates the second proof rule. It recurs on n+1, which is a bigger-sized problem: farther away from -not closer to- the base case.
- Although mathematically (n+1)!/(n+1) = (n+1)*n!//(n+1) = n! this function will continue calling factorial with ever-larger arguments: a runaway (or infinite) recursion. Actually, each recursive call can take up some space (to store its argument, see the hand simulation, which requires binding an argument for each recursive call), so eventually memory will be exhausted and Python will raise an exception.

Proof Rules

- Actually, Python limits the the number of times any recursive function can call itself. We can examine/set the recursion limit by importing the sys module and the calling sys.getrecursionlimit()/sys.setrecursionlimit(some number) functions.

```
def factorial (n):
    if n == 0:
        return 1
    else:
        return n+factorial(n-1)   # n+(n-1)! is not n!
```

Proof Rules

- This factorial function violates the third proof rule. Even if we assume that factorial(n-1) computes the correct answer, this function returns n added (not multiplied) by that value, so it does not return the correct answer. In fact, it returns one more than the sum of all the integer from 1 to n (because for 0 it returns 1) not the product of these numbers.
- In summary, each of these functions violates a proof rule and therefore doesn't always return the correct value. The first function always returns the wrong value; the second function returns the correct value, but only for the base case; it never returns a value for any other argument; the third function returns the correct value, but only for the base case.

Proving the Proof Rules

LECTURE 5

Proving the Proof Rules

- We can actually prove that these proof rules are correct! Here is the proof.
- This is not simple, but it is short so I will write the proof here and let you think about it (and reread it a dozen times if you need to).
- Assume that we have correctly proven that these three proof rules are correct for some recursive function `f`. And assume that we assert that the function is not correct. We will show that these two assertions lead to a contradiction.
- First, if `f` is not correct, then there must be some problems that it does not correctly solve. And, if there are any problems that `f` does not solve correctly, there must be a SMALLEST problem that it does not solve correctly: call this problem `p`.

Proving the Proof Rules

- Because of proof rule (1) we know that p cannot be the base case, because we have proven `f` recognizes and solves base cases correctly. So, f must solve `p` by recursion. Since `f` solves p by recursion, it first recursively solves a problem smaller than `p`: we know by proof rule (2) that it always recurs on a smaller problem size; and we know that `f` correctly solves this smaller problem, because `p`, by definition, is the smallest problem that `f` solves incorrectly. But we also know by proof (3) that assuming `f` solves all problems smaller than `p` (which it does, because `p` is the smallest problem `f` does not solve correctly), then `f` will use these solutions of smaller problems to solve the problem `p` correctly.
- So, `f` must solve `p` correctly, contradicting our assumption.
- Therefore, it is impossible to find a smallest problem that `f` incorrectly solves; so, `f` must solve all problems correctly.
- Well, that is how the proof goes.

Mathematics Recursively

LECTURE 6

Mathematics Recursively

- We can construct all the mathematical and relational operators on natural numbers (integers >= 0) given just three functions and if/recursion. We can recursively define the natural numbers as:

  0 is the smallest natural number

  for any natural number n, s(n) (the successor of n: n+1) is a natural number

Mathematics Recursively

- Now we define three simple functions z(ero), p(redecessor), and s(uccessor).

```
def z(n): # z(n) returns whether or not n is 0
    return n == 0
def s(n): # s(n) returns the successor to n (n+1)
    return n+1
def p(n): # p(n) returns the predecessor of n, if one exists
    if not z(n): # 0 has no predecessor
        return n-1
    else:
        raise ValueError('z: cannot compute predecessor of 0')
```

Mathematics Recursively

- Note we should be able to prove/argue/understand the following:

```
z(s(n)) is always False

p(s(n)) is always n

s(p(n)) is n if n != 0 (otherwise p(n) raises an exception)
```

- Given these functions, we can define functions for all arithmetic (+ - * // **) and relational (== <... and all the other relational) operators. For example

```
def sum(a,b):
    if z(a):                          # a == 0
        return b                      # return b: 0 + b = b
    else:                             # a != 0
        returns sum( p(a), s(b) )     # return (a-1)+(b+1) = a+b
```

Proof of correctness

1. The base case is a==0; and according to our knowledge of of mathematics, sum(0,b) is 0+b which is b. This function returns b when the argument a == 0.
2. If z(a) is not True (a is not 0), then p(a) as the first argument in the recursive call to sum is closer to the base case of 0. Because a is not 0, there is a predecessor of (a number one smaller than) a.
3. Assuming that sum(p(a),s(b)) computes its sum correctly, we have sum(p(a),s(b)) = (a-1)+(b+1) = a + b = sum(a,b), so returning this result correctly returns the sum of a and b.

Proof of correctness

- Another way to define this function is

```
def sum(a,b):
    if z(a):                          # a == 0
        return b                      # return b: 0 + b = b
    else:                             # a != 0
        returns s(sum( p(a), b )      # return (a-1)+(b) + 1 = a+b
```

- We can also use the 3 proof-rule to prove this function correctly computes the sum of any two non-negative integers.

Proof of correctness

- Given the sum function, we can similarly define the mult function, multiplying by repeated addition.

```
def mult(a,b):
    if z(a):                              # a = 0
        return 0                          # return 0: 0*b = 0
    else:                                 # a != 0
        return sum(b, mult(p(a),b))# return b+((a-1)*b)
                                          #        =b+a*b-b = a*b
```

Proof of correctness

- Switching from arithmetic to relational operators....

```
def equal(a,b):
    if z(a) or z(b):     # a = 0 or b = 0 (either == 0)
        return z(a) and z(b)# return True(if both == 0), False(if one != 0)
    else:                        # a != 0 and b != 0
        return equal(p(a),p(b))# return a-1==b-1 which is the same as a==b
```

- We also might find it useful to do a hand simulation of these functions, with the two parameters a and b stored in each "apartment" and passed as arguments
- The right way to illustrate all this mathematics is to write a class Natural, with these methods, and then overload/define __add__ etc. for all the operators.

Synthesizing recursive string methods

LECTURE 7

Synthesizing recursive string methods

We can define strings recursively:

'' is the smallest string

a character concatenated to the front of any string is a string

Synthesizing recursive string methods

1) Find the base (non-decomposable) case(s). Write the code that detects the base case and returns the correct answer for it, without using recursion
2) Assume that we can decompose all non base-case problems and then solve these smaller subproblems via recursion. Choose (requires some ingenuity) the decomposition; it should be "natural"
3) Write code that combines these solved subproblems (often there is just one) to solve the problem specified by the parameter

Synthesizing recursive string methods

We can use these rules to synthesize a method that reverses a string. We start with

```
def reverse(s):
```

(1) Please take time to think about the base case: the smallest string. Most students will think that a single-character string is the smallest, when in fact a zero-character string (the empty string) is smallest. It has been my experience that more students screw-up on the base case than the recursive case. Once we know the smallest string is the empty string, we need to detect it and return the correct result without recursion: the reverse of an empty string is an empty string.

```
def reverse(s):
    if s == '':          # or len(s) == 0
        return ''
    else:
        Recur to solve a smaller problem
        Use the solution of the smaller problem
        to solve the original problem
```

Synthesizing recursive string methods

- We can guess the form of the recursion as reverse(s[1:]) note that the slice s[1:] computes a string with all characters but the one at index 0: all the characters after the first. We are guaranteed to be slicing only on non-empty strings (those whose answer is not computed by the base case), so slicing will always be a smaller string: smaller by one character. We get to assume that the recursive call correctly returns the reverse of the string that contains all characters but the first.

```
def reverse(s):
    if s == '':          # or len(s) == 0
        return ''
    else:
        Use the solution of reverse(s[1:]) to solve the original
        problem
```

Synthesizing recursive string methods

- Now, think about an example. if we called reverse('abcd') we get to assume that the recursive call works: so reverse(s[1:]) is reverse('bcd') which we get to assume returns 'dcb'). How do we use the solution of this subproblem to solve the original problem, which must return 'dcba'? We need to concatenate 'a' (the first character, at s[0]) to the end of the reverse of all the other characters: 'dcb' + 'a', which evaluates to 'dbca', the reverse of all the characters in the parameter string. Generally, we write this function as

```
def reverse(s):
    if s == '':  # or len(s) == 0
        return ''
    else
        return reverse(s[1:]) + s[0]
```

Synthesizing recursive string methods

We have now written this method by ensuring the three proof rules are satisfied so we don't have to prove them, but note that

1) the reverse of the smallest string (empty) is computed/returned correctly
2) the recursive call is on a string argument smaller than s (all the characters from index 1 to the end, skipping the character at index 0, and therefore a string with one fewer characters)
3) ASSUMING THE RECURSIVE CALL RETURNS THE CORRECT ANSWER FOR THE SMALLER STRING, then by concatenating the first character after the end of it, we have correctly reversed the entire string (solving the problem for the original parameter).

Synthesizing recursive string methods

- In fact, we can use a conditional expression to rewrite this code as a single line as well.

```
def reverse(s):
    return ('' if s == '' else reverse(s[1:]) + s[0])
```

- Here is a similar recursive function for reversing the values in a list.

```
def reverse(l):
    if l == []:                          # or len(l) == 0
        return [];
    else
        return reverse(l[1:])+[l[0]] # [l[0]] for right operand of +
```

Synthesizing recursive string methods

- Now we will write a recursive function that returns the string equivalent of an int using the same approach: satisfying the three proof rules. We know that Python's str function, automatically imported from the builtins module) will return the string representation of an int. We can actually now write this function recursively and at the same time prove it is correct.
- To start, we assume that the integer is non-negative (and fix this assumption later). Unlike the factorial and power functions, here the size of the integer will be the number of digits it contains: the smallest non-negative integers (0-9) contain 1 digit, so that is the smallest size problem. So, we start with the header and base case.

Synthesizing recursive string methods

```
def to_str(n):
    if 0 <= n <= 9:
        return '0123456789'[n]                    # 0<=n<=9, so no
index error
    else:
        Recur to solve smaller problem s          # n has at least
two digits
        Use the solution of the smaller problems to solve
        the original problem
```

- We can guess the form of the recursion as to_str(n//10) and to_str(n%10) because n//10 is all but the last digit in n, and n%10 is the last digit. If n has at least d digits (where d>=2), then both n//10 and n%10 will have fewer digits: n//10 has d-1 digits and n%10 has 1 digit.

Synthesizing recursive string methods

- We get to assume that the recursive call correctly returns the string representation of these numbers.

```
def to_str(n):
    if 0<= n <= 9:
        return '0123456789’[n] # 0<=n<=9, so no index error
    else:
        Use the solution of to_str(n//10) and to_str(n%10)
```

Synthesizing recursive string methods

Now, think about an example. if we called to_str(135) we get to assume that the recursive calls work: so to_str(n//10) is to_str(13) which we get to assume it returns '13'; and to_str(n%10) is to_str(5) which we get to assume returns '5'. How do we use the solution of these subproblems to solve the original problem? We need to concatenate them togther. Generally we write this function as

```
def to_str(n):
    if 0<= n <= 9:
        return '0123456789’[n]    # 0<=n<=9, so no index error
    else:
        return to_str(n//10) + to_str(n%10)
```

Synthesizing recursive string methods

We have now written this method by ensuring the three proof rules are satisfied. Note that

1) the to_str of the smallest ints (1 digit) are computed/returned correctly
2) the two recursive calls are on int arguments that are smaller than n by at least one digit (in fact the second call is always exactly 1 digit).
3) ASSUMING THE RECURSIVE CALLS WORK CORRECTLY FOR THE SMALLER int , then by concatenating the two numbers together, we have correctly found the string representation of the n (solving the original problem)

Synthesizing recursive string methods

- We make this function work for negative numbers by redefining to_str with its original code in a locally defined function, changing the body of this function by either appending nothing or a '-' in front of the answer, depending on n.

```
def to_str(n):
    def to_str1(n):                    # n >= 0 (see call with abs)
        if 0<= n <= 9:
            return '0123456789’[n]  # 0<=n<=9, so no index error
        else:
            return to_str1(n//10) + to_str1(n%10)
    return ('' if n >= 0 else '-')+to_str1(abs(n))
    # or
    #return (to_str1(n) if n >= 0 else '-'+to_str1(-n))
```

Synthesizing recursive string methods

- In fact, the following function uses the same technique (but generalizes it by converting to an arbitrary base) to compute the string representation of a number in any base from binary up to hexadecimal: to_str(11,2) returns '1011'

```
def to_str(n,base=10):                  # bases 2 - 16
    if 0<= n <= base-1:
        return '0123456789ABCDEF'[n]# 0<=n<=15, so no index error
    else:
        return to_str(n//base,base) + to_str(n%base,base)
```

Synthesizing recursive string methods

- Now let's write a method that has two recursive parameters. Suppose we want to write a same_length function that tests whether the length of its two string parameters are equal, without ever explicitly computing the length of each.
- With two recursive parameters we have possible 4 base conditions

		Parameter 2	
		Empty	Not empty
Parameter 1	Empty	Equal	Not equal
	Not Empty	Not Equal	recur

Synthesizing recursive string methods

- In three of the four, we immediately know the answer. If both parameters are empty then the strings have the same length; if one parameter is empty and one isn't, then the strings have different lengths. Only if both are not empty do we need to recur to compute the correct answer.

Synthesizing recursive string methods

- Here are three ways to write the base cases.

```
if s1 == '' and s2 == '':
    return True
if s1 == '' and s2 != '':
    return False
if s1 != '' and s2 == '':
    return False;

if s1 == '':
    return s2 == ''
if s2 == ''
    return False                    # if got here, s1 != ''

if s1 == '' or s2 == '':            # if either is empty, will return
    return s1 == '' and s2 == ''    # return True if both empty

if s1 == '' or s2 == '':            # if either is empty, will return
    return s1 == s2                 # return True if the same (empty)
```

Synthesizing recursive string methods

- So, we can start this function as

```
def same_length(s1,s2)
    if s1 == '' or s2 == '':
        return s1 == '' and s2 == ''
    else:
        Recur to solve a smaller problem
        # s1/s2 each are not empty
        Use the solution of the smaller problem to
        solve the original problem
```

Synthesizing recursive string methods

- Now, if Python executes the else: clause then it has two non-empty strings, for each of which we can compute a substring (all the characters after the first). If the substrings have the same length, then the original strings have the same length; if the substrings don't have the same length then the original strings don't have the same length. So, solving this problem for the substrings is exactly the same as solving it for the original strings. So we can write the recursive call as

```
def same_length(s1,s2):
    if s1 == '' or s2 == '':
        return s1 == '' and s2 == ''
    else:
        return same_length(s1[1:],s2[1:])
```

Note that if we compared the lengths of a huge string and a tiny one, we would find that they are different in an amount of time proportional to the tiny string.

Recursive list processing

LECTURE 8

Recursive list processing

- Finally, here are some some simple recursive list processing functions. As with strings, we can slice a list to get a smaller list, with the slice l[1:] especially common and useful.
- Could you start from scratch and define this as illustrated above?
- We can define lists recursively:

  [] is a list

  a value concatenated to the front of a list is a list

Recursive list processing

- If there were not len function for lists, we could easily define it recursively as

```
def len(l):
    if l == []:
        return 0
    else:
        return 1 + len(l[1:])
```

- Likewise for a sum function

```
def sum(l):
    if l == []:
        return 0
    else:
        return l[0] + sum(l[1:])
```

Recursive list processing

- Below, the all_pred function returns True if and only if predicate p always returns True (never returns False), when called on every value in the list.

```
def all_pred(l,p): # where p is some predicate whose
                   # domain includes l's values
    if l == []:
        return True
    else:
        return p(l[0]) and all_pred(l[1:],p)
```

Recursive list processing

- Note that because and is a short-circuit operator, it recurs only as far as the first False value, at which point it does not need to call `all_pred(l[1:])` recursively. When we study efficiency, we will discover that the way Python represents lists (as growable arrays) make recursion inefficient compared to iteration, but when we study linked list and trees (briefly this quarter, extensively in ICS-46) we will see for those implementations, recursion is as fast at iteration.

Recursive list processing

- Finally, you might wonder why the base case, `all_pred([],p)` returns True. What should the function return for an empty list? Well, imagine we are one call before the empty list: a list with one value. What should `all_pred([a],p)` return. Well, it should return p(a)
- True for this one-element list. What does the recursive part of this function return:

```
p(a) and all_pred([],p).
```

- So, we need to solve the equation by determing what all_pred([],p) should be.

```
p(a) == p(a) and all_pred([],p)
```

Recursive list processing

- To solve this equation, and determine the value of all_pred([],p), we find that all_pred([],p) must be True: if it were False, p(a) and all_pred([],p) would be the same as p(a) and False, which would always be False, not the required answer of p(a).
- Based on this same logic, here are what based cases must be, categorized by the operator before the recursive call.

```
base case = True  ...  and recursive-call (as we saw in all)
base case = False ...  or  recursive-call
base case = 0     ...  +   recursive-call
base case = 1     ...  *   recursive-call (as we saw in ! and **)
base case = -infinity ... max(...,recursive_call)
base case = +infinity ... min(...,recursive_call)
```

- Generally, x op recursive-call(base case) must be x, which it is for all these values. and operators.