Recursive Programs and Python

COMPSCI 230 — Discrete Math

January 19, 2016
Outline

1. A Running Example: mergeSort
2. The Recursive Helper and Recursive Thinking
3. Functions
4. Aliasing, Mutables, and Side Effects
5. Anonymous Functions
6. Sequence Slices
7. Back to Recursion
8. Completing mergeSort
An Example: mergeSort

- Sort the sequence (a list or a string) 
  \[2, 6, 3, 8, 1, 5, 6\]
- Need to specify an ordering criterion:
  - \(3 < 5\)
  - \('d' < 'h'\)
  - "apple" < "application"
  - "June 17, 2014" < "May 8, 2016"
  - "hut" < "house"?
  - ...
mergeSort with a Helper

• Sort the sequence [2, 6, 3, 8, 1, 5, 6]
• Think of a helper who can do smaller problems, except the very smallest
• Very small problems: [ ] or [5] or [8, 1]
• When is small “the very smallest?”
• We’ll get back to that
mergeSort with a Helper

• Splitting the problem into smaller ones requires creativity
• Different split criterion, different algorithm
• “About halfway:”
  • Split \([2, 6, 3, 8, 1, 5, 6]\) into
    \(L = [2, 6, 3]\) and \(R = [8, 1, 5, 6]\)
  • Ask the helper to sort \(L\) and \(R\)
  • Get back
    \(L = [2, 3, 6]\) and \(R = [1, 5, 6, 8]\)
• Wait, who writes the code for the helper?
• Relax! This is recursive thinking
The Smallest Problem(s)

• Very small problems: [ ] or [5] or [8, 1]
• “The very smallest” means: all the cases you cannot shrink further:
  • A sequence of two items can be shrunk into two of one item each (2 is not “smallest”)
  • A sequence of one item cannot be shrunk into shorter sequences (1 is “smallest”)
  • An empty sequence cannot be shrunk into shorter sequences (0 is “smallest”)
  • Lengths 0 and 1 are “the very smallest”

• These are called base cases
• Need to think about base cases carefully for different problems
**mergeSort with a Helper, Ct’d**

- \( L = [2, 3, 6] \), \( R = [1, 5, 6, 8] \)
- Can merge \( L \) and \( R \) by repeatedly comparing their first elements, and picking the smaller of the two
  
  \[
  \begin{align*}
  L &= [2, 3, 6] \quad R &= [1, 5, 6, 8] \quad S &= [] \\
  L &= [2, 3, 6] \quad R &= [5, 6, 8] \quad S &= [1] \\
  L &= [3, 6] \quad R &= [5, 6, 8] \quad S &= [1, 2] \\
  L &= [6] \quad R &= [5, 6, 8] \quad S &= [1, 2, 3] \\
  L &= [6] \quad R &= [6, 8] \quad S &= [1, 2, 3, 5] \\
  L &= [] \quad R &= [6, 8] \quad S &= [1, 2, 3, 5, 6] \\
  
  L \text{ is empty, just empty out } R \text{ into } S: \quad L = [] \quad R = [8] \quad S = [1, 2, 3, 5, 6, 6] \\
  L = [] \quad R = [] \quad S = [0, 1, 2, 3, 5, 6, 6, 8]
  \end{align*}
\]
mergeSort Code Development

• We are done! Let’s write the code
• Wait, who writes the code for the helper?
• Sshhh! Relax! Think recursively!
• Let’s write comments first, then translate to code
• So we separate thinking about *what* we do...
• ... from *how* we do it in Python
Comments First

# a function that sorts seq with criterion before
# if there are at least two items in the list
# mid is half the length of seq
# left is the left half of seq
# right is the right half of seq
# ask the helper to sort left
# ask the helper to sort right
# merge left and right into seq

We’ll flesh out the red part later
Using indentation to delineate blocks. Python does, too
Functions

# a function that sorts seq with criterion before
def mergeSort(seq, before):
    ... (indented)

- We use camelCase for names, but you can use underscores_instead or another convention
- A colon starts a block, which follows further indented
- Exception: you could put a single instruction (with no additional colons) on the same line
- The argument before is the name of a function: functions are (almost) first-class citizens in Python
Aliasing and Side Effects

def mergeSort(seq, before):
    ... (body of mergeSort)
def greater(a, b): return a > b  # for non-increasing order

The instructions

s = [2, 1, 4, 1, 6]
mergeSort(s, greater)

change s to [6, 4, 2, 1, 1]

• s and seq are aliases for each other
• They refer to the same object
• mergeSort has side effects: it alters s
• Different from Matlab or Java!
Aliasing and Mutable Objects

- Aliasing in Python occurs only for *mutable objects*
- For our purposes
  - Numbers and strings are immutable
  - Lists (and dictionaries) are mutable

```python
>>> list1 = ['a', 'b', 'c']
>>> list2 = list1
>>> list1 += [1, 2, 3]
>>> list1
['a', 'b', 'c', 1, 2, 3]
>>> list1
['a', 'b', 'c', 1, 2, 3]  # list1 and list2 are aliases
```
Anonymous Functions

def greater(a, b): return a > b  # decreasing order
s = [2, 1, 4, 1, 6]
mergeSort(s, greater)

• changes s to [6, 4, 2, 1, 1]
• more concise syntax:

s = [2, 1, 4, 1, 6]
mergeSort(s, lambda a, b: a > b)

• *Anonymous function*
• No argument parentheses
• No return
• Body of *lambda* can only be a simple expression
Default Arguments

# a function that sorts seq with criterion before
def mergeSort(seq, before):

- More often than not, we want to:
  mergeSort(s, lambda a, b: a < b))

- Make before a default argument in the def:
  def mergeSort(seq, before = lambda a, b: a < b)

- So then we can call
  mergeSort(s)
  unless we want a different ordering criterion

- Python also provides for keyword arguments

- You can program well without them
Sequence Slices

def mergeSort(seq, before = lambda a, b: a < b):
    if len(seq) > 1:
        mid = len(seq) // 2
        left = seq[:mid]
        right = seq[mid:]
        # some more stuff...

• \texttt{len} is a built-in function
• \texttt{//} is integer division
• \texttt{Slice} \texttt{s[a:b]} of sequence \texttt{s} is \texttt{[s[a], s[a+1], \ldots s[b-1]]}
• Sequence indices start at 0
• \texttt{a} defaults to 0 and \texttt{b} defaults to \texttt{len(s)}
• Can specify a \textit{stride}: \texttt{s[a:b:c]} (every \texttt{c}-th element)
• Stride can be negative: start at \texttt{a} (greater than \texttt{a}), end just above \texttt{b}
• Stride is the third element, not the second (as in Matlab)!
Recursive Calls

def mergeSort(seq, before = lambda a, b: a < b):
    if len(seq) > 1:
        mid = len(seq) // 2
        left = seq[:mid]
        right = seq[mid:]
        mergeSort(left, before)
        mergeSort(right, before)
        # merge left and right into seq

• Just call the function itself!
• Need to specify any default arguments explicitly, because the ”helper” needs to know what the caller wants
• *Lots* of ”helper calls”: top calls two helpers, each of them calls two more, ...
• The number of ”helpers” grows exponentially, but the lengths of the sequences shrink exponentially
Back to Recursion

Calls and Returns

Recursive calls

No work and return

Merge and return

[8, 3, 1, 5, 6, 4, 3, 2, 6]
[8, 3, 1, 5]
[6, 4, 3, 2, 6]
[8, 3]
[1, 5]
[6, 4]
[3, 2, 6]
[8]
[3]
[1]
[5]
[6]
[4]
[3]
[2, 6]
[3, 8]
[1, 5]
[4, 6]
[2, 3, 6]
[1, 3, 5, 8]
[1, 5]
[4, 6]
[2, 3, 4, 6, 6]
[1, 2, 3, 3, 4, 5, 6, 6, 8]
def mergeSort(seq, before = lambda a, b: a < b):
    '''Sort the list seq by the comparison criterion cmp (default is "<")'''
    if len(seq) > 1:
        mid = len(seq) // 2
        left = seq[:mid]
        right = seq[mid:]
        mergeSort(left, before)
        mergeSort(right, before)
        (i, j, k) = (0, 0, 0)
        while i < len(left) and j < len(right):
            if before(left[i], right[j]):
                seq[k] = left[i]
                i += 1
            else:
                seq[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            seq[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            seq[k] = right[j]
            j += 1
            k += 1

• Blue: split sequence and call recursively
• Red: merge left and right into seq and return
• Green: did you hit a base case?
• Tuples like (i, j, k) abbreviate multiple assignments
• Logical connectives are English words: and, or, ...
• The DocString at the top needs triple quotes and can be retrieved as mergeSort.__doc__
• i += 1 abbreviates i = i + 1 (works with all binary ops)