Recursive Programs and Python

COMPSCI 230 — Discrete Math

January 17, 2017
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 list
  \[2, 6, 3, 8, 1, 5, 6]\]
- Need to specify an ordering criterion:
  - \(3 < 5\)
  - \(3 > 5\)
  - 'd' < 'h'
  - "apple" < "application"
  - "June 17, 2014" < "May 8, 2017"
  - "hut" < "house"
  - ...
- Specify a function before to pass to the sorter
mergeSort with a Helper

- Sort the list $[2, 6, 3, 8, 1, 5, 6]$
- 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! You don’t want to know
- 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 list of two items can be shrunk into two of one item each (2 items is not “smallest”)
  - A list of one item cannot be shrunk into shorter lists (1 item is “smallest”)
  - An empty list cannot be shrunk into shorter lists (0 items is “smallest”)
  - Lengths 0 and 1 are “the very smallest”

- These are called base cases
- They depend on the split criterion
- Getting base cases right is half of the work
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

  \[
  \begin{align*}
  &L = [2, 3, 6] \quad R = [1, 5, 6, 8] \quad S = [] \\
  &L = [2, 3, 6] \quad R = [5, 6, 8] \quad S = [1] \\
  &L = [3, 6] \quad R = [5, 6, 8] \quad S = [1, 2] \\
  &L = [6] \quad R = [5, 6, 8] \quad S = [1, 2, 3] \\
  &L = [6] \quad R = [6, 8] \quad S = [1, 2, 3, 5] \\
  &L = [] \quad R = [6, 8] \quad S = [1, 2, 3, 5, 6] \\
  \end{align*}
  \]

- \( L \) is empty, just empty out \( R \) into \( S \):

  \[
  \begin{align*}
  &L = [] \quad R = [8] \quad S = [1, 2, 3, 5, 6, 6] \\
  &L = [] \quad R = [] \quad S = [0, 1, 2, 3, 5, 6, 6, 8] \\
  \end{align*}
  \]
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 that in Python
# Recursive Helper and Recursive Thinking

Comments First

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

We sort lst *in place* (no copies, no return)
Using *indentation* to delineate blocks.
Python does, too
We’ll flesh out the red part later
Functions

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

- 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(lst, 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 lst are \textit{aliases} for each other

• They refer to the same object

• mergeSort has \textit{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, strings, and tuples 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]
>>> list2
['a', 'b', 'c', 1, 2, 3]  # list1 and list2 are aliases
```
Anonymous Functions

def greater(a, b): return a > b  # non-increasing 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
Anonymous Functions

Default Arguments

# a function that sorts lst with criterion before
def mergeSort(lst, 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(lst, before = lambda a, b: a < b)

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

• Then you would say something like
  mergeSort(s, lambda a, b: a > b))
  or mergeSort(s, before=lambda a, b: a > b))

• The latter *named parameter* is useful when there are several default arguments
def mergeSort(lst, before = lambda a, b: a < b):
    if len(lst) > 1:
        mid = len(lst) // 2
        left = lst[:mid]
        right = lst[mid:]
        # some more stuff...

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

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

• 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 ”helper calls” grows exponentially, but the lengths of their lst arguments shrink exponentially
Calls and Returns

[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]

[3, 8]
[1, 5]
[4, 6]
[2, 3, 6]

[3, 1, 5, 8]
[1, 3, 5, 8]
[2, 3, 4, 6, 6]
[1, 2, 3, 3, 4, 5, 6, 6, 8]
def mergeSort(lst, before=lambda a, b: a < b):
    """Sort the list lst by the comparison criterion before (default is ",")""
    if len(lst) > 1:
        mid = len(lst) // 2
        left = lst[:mid]
        right = lst[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]):
                lst[k] = left[i]
                i += 1
            else:
                lst[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            lst[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            lst[k] = right[j]
            j += 1
            k += 1
    • **Blue**: split list and call recursively
    • **Red**: merge left and right into lst
    • **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)
A Subtlety

After exhausting the shorter of left and right in the merge, we did

```python
while i < len(left):
    lst[k] = left[i]
    i += 1
    k += 1

while j < len(right):
    lst[k] = right[j]
    j += 1
    k += 1
```

Can we replace this with the following, shorter code?

```python
lst = lst[:k] + left[i:] + right[j:]
```

No: the + creates a copy of lst.
We want to modify lst itself ("in place"):

```python
del lst[k:]
lst.extend(left[i:])
lst.extend(right[j:])
```