CSC148 winter 2016

reading recursion week 6

Danny Heap heap@cs.toronto.edu BA4270 (behind elevators)

http://www.cdf.toronto.edu/~csc148h/winter/ 416-978-5899

February 26, 2016





Outline

test #1 follow-up

recursion on nested lists

recursion with turtles



announcements

- ▶ office hours, Monday/Wednesday/Thursday 3-5, BA4270 OR BA2230
- ▶ also CS help centre Wednesday and Thursday 4-6



length, inter-section comparison

- ▶ mean of the 10 a.m. and 1 p.m. was a statistical tie, 6 p.m. was lower
- ▶ 6 p.m. mean is adjusted 0.5/30 higher
- overall average 67.9%
- ▶ there was a lot of writing, see the proposal several slides on



post-test exercise

- ▶ 1% post-test exercise, follow instructions on sticker, either on last page of test paper, or an inner page if there is no room
- exercise is due March 2nd, 11:59 p.m., not the date on the sticker!
- ▶ in testing there was an occasional error in submitting, which is fixed by reloading the web page and continuing



fire alarm incident

- ▶ about 180 students had their test interrupted by a fire alarm
- our marking scheme has no provision for make-up tests; all likely dates overlap things such as assignment due dates or other course events
- ▶ for individuals who miss a test for valid reason, we re-evaluate the mark based on the second test and final
- ► consulting our department's undergraduate chair, we use a formula we believe neither gives an advantage nor a disadvantage to the affected students (see next slide)

replace test #1 grade

 a_1 : class average on test #1

 a_2 : class average on test #2

 a_e : class average on on final exam

 g_2 : student's grade on test #2

 g_e : student's grade on final exam

test #1 score:
$$g_1=rac{g_2/a_2+g_e/a_e}{2} imes a_1$$

rationale: student standing the same compared to the average on test #1 as compared to the average on test #2 and the final



what about those who didn't have a fire alarm?

although we think the formula for those who missed test #1 gives them neither an advantage nor a disadvantage, we will offer the remaining students the maximum of either their current grade on test #1 or the grade calculated using the formula on the previous slide

if a majority of students vote for this change, students who perform better relative to their peers on test #2 and the final may improve their test#1 grade

the vote will be in class, on March 2nd





summing lists

```
L1 = [1, 9, 8, 15]

sum(L1) = ???

L2 = [[1, 5], [9, 8], [1, 2, 3, 4]]

sum([sum(row) for row in L2]) = ??

L3 = [[1, 5], 9, [8, [1, 2], 3, 4]]
```

How can we sum L3?

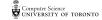


re-use built-in... recursion!

▶ a function sum_list that adds all the numbers in a nested list shouldn't ignore built-in sum

- ...except sum wouldn't work properly on the nested lists, so make a list-comprehension of their sum_lists
- but wait, some of the list elements are numbers, not lists!

write a definition of sum_list — don't look at next slide yet!





hey! don't peek!

```
def sum_list(L):
    ''', (list or int) -> int
    Return L if it's an int, or sum of the numbers in possibly nested 1
    >>> sum_list(17)
    17
    >>> sum_list([1, 2, 3])
    6
    >>> sum_list([1, [2, 3, [4]], 5])
    15
    , , ,
    # reuse: isinstance, sum, sum_list !
    if isinstance(L, list):
        return sum([sum_list(x) for x in L])
    else: # L is an int
        return I.
```

tracing recursion

To understand recursion, trace from simple to complex:

- ▶ trace sum_list(17)
- ▶ trace sum_list([1, 2, 3]). Remember how the built-in sum works...
- ▶ trace sum_list([1, [2, 3], 4, [5, 6]]). Immediately replace calls you've already traced (or traced something equivalent) by their value
- ▶ trace sum_list([1, [2, [3,4], 5], 6 [7, 8]]). Immediately replace calls you've already traced by their value.





depth of a list

Define the depth of L as 1 plus the maximum depth of L's elements if L is a list, otherwise 0.

- ▶ the definition is almost exactly the Python code you write!
- start by writing return and pythonese for the definition:

```
if instance(L, list):
    return 1 + max([depth(x) for x in L])
else: # L is not a list
    return 0
# find the bug! (then fix it...)
```

▶ deal with the special case of a non-list





trace to understand recursion

Trace in increasing complexity; at each step fill in values for recursive calls that have (basically) already been traced

- Trace depth([])
- ▶ Trace depth(17)
- Trace depth([3, 17, 1])
- ▶ Trace depth([5, [3, 17, 1], [2, 4], 6])
- ► Trace depth([14, 7, [5, [3, 17, 1], [2, 4], 6], 9])

maximum number in nested list

Use the built-in max much like sum

- how would you find the max of non-nested list? max(...)
- how would you build that list using a comprehension? max([...])
- what should you do with list items that were themselves lists?

```
max([rec_max(x) ...])
```

▶ get some intuition by tracing through flat lists, lists nested one deep, then two deep...





code for rec_max

```
if isinstance(L, list):
    return max([rec_max(x) for x in L])
else:
    return L
```

trace the recursion

trace from simple to complex; fill in already-solved recursive calls

▶ trace rec_max([3, 5, 1, 3, 4, 7])

▶ trace rec_max([4, 2, [3, 5, 1, 3, 4, 7], 8])

▶ trace
 rec_max([6, [4, 2, [3, 5, 1, 3, 4, 7], 8], 5])



get some turtles to draw

Spawn some turtles, point them in different directions, get them to draw a little and then spawn again...

Try out tree_burst.py

Notice that tree_burst returns NoneType: we use it for its side-effect (drawing on a canvas) rather than returning some value.



nested_contains

Return whether a list, or any of its sublists, contain some non-list value.

- ▶ should return True if any element is equivalent to value
- ▶ should return True if any element is a list ultimately containing value
- Python any and functional if are useful

<expression 1> if <condition> else <expression 2>

If the condition is true, evaluates to the first expression, otherwise evaluates to the second expression.



base case, general case

You will have noticed that a recursive function has a conditional structure that specifies how to combine recursive subcalls (general case), and when/how to stop (the base case, or cases).

What happens if you leave out the base case?

