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
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.
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

\[ g_1 = \frac{g_2/a_2 + g_e/a_e}{2} \times a_1 \]

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!
def sum_list(L):
    ''' (list or int) -> int

    Return L if it’s an int, or sum of the numbers in possibly nested list L
    
    >>> sum_list(17)
    17
    >>> sum_list([1, 2, 3])
    6
    >>> sum_list([1, [2, 3, [4]], 5])
    15
    '''

    if isinstance(L, list):
        return sum([sum_list(x) for x in L])
    else: # L is an int
        return L
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.
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:

  ```python
  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
	race from simple to complex; fill in already-solved recursive calls

- trace \text{rec\_max}([3, 5, 1, 3, 4, 7])

- trace \text{rec\_max}([4, 2, [3, 5, 1, 3, 4, 7], 8])

- trace

  \text{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.
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.
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?