Outline

recursion on nested lists

recursion with turtles
def distribute(handouts, raisedHands):
    if len(raisedHands) == 0 or len(handouts) == 1:
        keep handouts
    elif len(handouts) == 2:
        take 1 handout
        neighbour does distribute(1 handout, raisedHands)
    else:
        take 1 handout, split remainder into handouts/2
        2 neighbours do distribute(handouts/2, raisedHands)
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?
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):
    
    Return the sum of all ints in L.

    @param int|list[int|list[...]] L: possibly-nested list of ints, finite depth

    >>> sum_list([1, [2, 3], [4, 5, [6, 7], 8]])
    36
    >>> sum([[]])
    0
    
    if isinstance(L, list):
        return sum([sum_list(x) for x in L])
    else:
        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 isinstance(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?
  \[ \text{max}(...) \]

- how would you build that list using a comprehension?
  \[ \text{max}([...]) \]

- what should you do with list items that were themselves lists?
  \[ \text{max}([\text{rec\_max}(x) \ldots]) \]

- get some intuition by tracing through flat lists, lists nested one deep, then two deep...
if isinstance(L, list):
    return max([rec_max(x) for x in L])
else:
    return L
trace the recursion

draw the recursion 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])
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?
template for structural recursion

recursion when input is a recursive structure:

- if input cannot be decomposed into recursive sub-structures, you have a base case and you directly return a result without recursion
- if input can be decomposed into recursive sub-structures, solve them recursively and combine the result(s)

this reduces your job to (a) figuring out how to detect whether the input can be decomposed or not, (b) figuring out how what result to return for the base case, and (c) figuring out which substructures to solve recursively and how to combine their solutions
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.