you will become experts in structural recursion...

CSC148 winter 2018

reading recursion

week 6

Danny Heap

heap@cs.toronto.edu

BA4270 (behind elevators)

http://www.teach.cs.toronto.edu/~csc148h/winter/

416-978-5899

February 12, 2018

Outline

recursion on nested lists

recursion with turtles

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]]
```

```
def sum1(L1):
    return sum(L1)

def sum2(L2):
    return([sum1(x)
            for x in L2])
```

??

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(list_: List[int]) -> int:
    """
    Return the sum of all ints in list_.

    >>> sum_list([1, [2, 3], [4, 5, [6, 7], 8]])
    36
    >>> sum([])
    0
    """

    if isinstance(list_, list):
        return sum([sum_list(x) for x in list_])
    else:
        return list_
```

— general case

— base case — no recursion

but wait: can you call a function before it's defined?

```
>>> def f(n):
...     return g(n) + 1
...
>>> f(2) # CRASH!
>>> def g(n):
...     return 2 * n
...
>>> f(2)
```

define time okay!

— now it works

tracing recursion

To understand recursion, trace from simple to complex:

See work sheet

- 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 list_ as 1 plus the maximum depth of list_'s elements if list_ 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 isinstance(list_, list):
    return 1 + max([depth(x) for x in list_])
else: # list_ 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([])`

  → 1 + max([depth(x) for x in []])

  → 1 + max([])

- Trace `depth(17)`

  → return 0 → 0.

- Trace `depth([3, 17, 1])`

  → 1 + max([depth(3), depth(17), depth(1)])

  → 1 + max([0, 0, 0]) → 1

- Trace `depth([5, [3, 17, 1], [2, 4], 6])`

  → 1 + max([d(5), d([3, 17, 1]), d([2, 4]), d(6)])

  → 1 + max([0, 1, 1, 0]) → 2

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

  → 1 + max([d(14), d(7), d([5, [3, 17, 1], [2, 4], 6]), d(9)])

  → 1 + max([0, 0, 2, 0]) → 3

base case, general case

s

no recursion

recursive calls

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 to do here?

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 write an if block (maybe elif)
- 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

→ master recursion (on recursive data structures)

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(list_, list):
    return max([rec_max(x) for x in list_])
else:
    return list_
```

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])`

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.

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.