CSC148 Lab#3, winter 2015

learning goals

In this lab you will:

- practice reading and tracing recursive functions guided by their docstring;
- practice reading and tracing recursive functions that have a minimal docstring;
- practice writing recursive functions, guided by a docstring with appropriate examples.

These exercises follow the techniques of our week # 3 lectures, which you may want to review.

You are encouraged to start working on this lab as soon as it is posted. If you feel shaky on the lab, be sure to come in and work through it with your TA. At the end of this handout there are additional exercises designed to increase your confidence. There will be a quiz during the last 15 minutes of the lab which you are likely to ace if you have worked through the lab.

reading and understanding recursion with a docstring

Each of the functions in this section have a complete docstring, however you may not believe that the functions work as advertised.

After each function definition there are some tracing exercises for you to carry out. For each function you must carry out the tracing **in order**.

1. Replace the function call with the function definition, after you have filled in the values passed as arguments, chosen the appropriate "branch" of the **if** statement, and unwrapped any list comprehensions. You may replace **return** by -->
2. Below this, re-write your filled-in definition by replacing any sub-calls to your recursive function with the value they return, if you have **already traced** an equivalent function call.

   The last tracing exercise for each function will be a special case where you will **not** have already traced a function call equivalent to your recursive sub-call. In these cases you will rely on the docstring to tell you what value to fill in.
3. Evaluate your filled-in definition to see what value is returned.

If this sounds mysterious, look over the **nested_concat** example below.

nested_concat

Python **lists** can be nested arbitrarily deeply:

```
[1, [2, 3, [4, 5], 6], [7, [8, [9, [10]]]]]
```

Indeed, even infinite nesting is possible:

```
>>> L = [1, 2]
>>> L.append(L) # !
```

...but we won't go there (today).

Solving problems for such nested lists often means solving the same problem for the sublists, and then combining those solutions. Recursion is your friend here. Read over the code that "sums" (concatenates) arbitrarily-nested list of **str**:

```
def nested_concat(L):
    '''(list or str) -> str

    Return L if it's a str, if L is a (possibly-nested) str return
    concatenation of str elements of L and (possibly-nested) sublists of L.

    Assume: Elements of L are either str or lists with elements that
            satisfy this same assumption.

    # examples omitted!
    '''
    if isinstance(L, str):
        return L
    else: # L is a possibly-nested list of str
        return ''.join([nested_concat(x) for x in L])
```

Now we trace the following calls **in order**. It is important to plug in a value when you see a recursive call you have already solved, **rather than tracing any further!**[1]

1. Trace **nested_concat('five')**

```
nested_concat('five') --> 'five' # since isinstance('five', str)
```

2. Trace **nested_concat([])**.

```
nested_concat([]) --> ''.join([nested_concat(x) for x in []])
   --> ''.join([]) # there are no x in []
   --> ''          # that's what join does to empty list
```

3. Trace **nested_concat(['how', 'now', 'brown'])**.

```
nested_concat(['how', 'now', 'brown'])
   --> ''.join([nested_concat(x) for x in ['how', 'now', 'brown']])
   --> ''.join(['how', 'now', 'brown']) # traced nested_concat(str) before
   --> 'hownowbrown' # join does 'how' + 'now' + 'brown'
```

4. Trace **nested_concat(['how', ['now', 'brown'], 'cow'])**

```
nested_concat([['how', ['now', 'brown'], 'cow']])
   --> ''.join([nested_concat(x) for x in ['how', ['now', 'brown'], 'cow'])
   --> ''.join(['how', 'nowbrown', 'cow']) # traced nested_concat(list of str)
                                           # and nested_concat(str) before
   --> 'hownowbrowncow'
```

5. trace **nested_concat([['how', ['now', 'brown', ['cow']], 'eh?']])**

```
nested_concat([['how', ['now', 'brown', ['cow']], 'eh?']])
   --> ''.join([nested_concat(x) for x in [['how', ['now', 'brown', ['cow']], 'eh?']]
   --> ''.join(['how', 'nowbrowncoweh?']) # believe docstring for str and possibly-nested
                                          # list of str or possibly-nested...
   --> 'hownowbrowncoweh?'
```

[1] Of course, you could secretly ignore this instruction and run the code through the Python visualizer. That would show you how a particular computing model implements recursion but **NOT** how humans understand recursion.

count non-list elements

If a list **L** doesn't contain sub-lists, the number of non-list elements it contains is simply **len(L)**. What happens if **L** might contain sublists, and those sublists might have their own sublists? Read over the code to count non-list elements:

```
def count_elements(L):
    '''(list or non-list) -> int

    Return 1 if L is a non-list, or number of non-list elements in
    possibly-nested list L.

    Assume: L is any Python object.

    # examples omitted!
    '''
    if isinstance(L, list):
        return sum([count_elements(x) for x in L])
    else: # if L is not a list, count it
        return 1
```

Now trace the following calls **in order** on some scrap paper. It is important to plug in a value when you see a recursive call you have already solved, **rather than tracing any further!**

1. Trace **count_elements('snork')**
2. Trace **count_elements([2, 1, 3])**
3. Trace **count_elements([[2, 1, 3, '[4]']])**
4. Trace **count_elements([7, [2, 1, 3], 9, [3, 2, 4]])**
5. Trace **count_elements([7, [2, [1, 3]], 9, [3, 2, 4]])**

After you've argued with your partner enough, call your TA over and show your work.

reading recursion with incomplete docstring

In this section we show how to trace a recursive function using only the function body — the docstring provides no support. Use the same tracing approach as in the previous section, but now you have no stated assumptions that you can use.

nd

Read the definition of **nd**. It doesn't have the purpose statement, any assumptions, or examples filled in!

```
def nd(L):
    '''(list or non-list) -> int
    '''
    if isinstance(L, list):
        return 1 + max([nd(x) for x in (L + ['ox'])])
    else: # L is a non-list
        return 0
```

Now trace the following calls on **nd**. It is important to trace these **in order**, and to fill in the value of any recursive sub-call to **nd** if you have already traced a similar call.

1. Trace **nd('ox')**

```
nd('ox') --> 0 # 'ox' is a non-list
```

2. Trace **nd([])**

```
nd([]) --> 1 + max([nd(x) for x in ['ox']]) # L is a list
--> 1 + max([0]) # already traced nd('ox')
--> 1 + 0 --> 1
```

3. Trace **[1, 2, 3]**

```
nd([1, 2, 3]) --> 1 + max([nd(x) for x in [1, 2, 3, 'ox']])
--> 1 + max([0, 0, 0, 0]) # already traced nd of non-list
--> 1 + 0 --> 1
```

4. Trace **nd([1, [2, 3], 4])**

```
nd([1, [2, 3], 4])--> 1 + max([nd(x) for x in [1, [2, 3], 4, 'ox']])
--> 1 + max([0, 1, 0, 0]) # already traced nd of non-list and list of non-list
--> 1 + 1 --> 2
```

5. Trace **nd([5, [[1, [2, 3], [4]]]])**

```
nd([5, [[1, [2, 3], [4]]]])
--> 1 + max([nd(x) for x in [5, [[1, [2, 3], [4]]], 'ox']])
--> 1 + max([0, 3, 0]) --> 1 + 3 --> 4
# believe nd(list) --> 1 + max nd of lists's elements
```

nm

Read the definition of **nm**. It doesn't have the purpose statement, any assumptions, or examples filled in!

```
def nm(L):
    ''' (list or int) -> int
    '''
    if isinstance(L, list):
        return max([nm(x) for x in L])
    else: # L is an int
        return L
```

Now trace the following calls on **nm**. It is important to trace these **in order**, and to fill in the value of any recursive sub-call to **nm** if you have already traced a similar call (you'll know that because you will have filled in something under "Assume..."):

1. Trace **nm(6)**
2. Trace **nm([1, 3, 7, 5, 2])**
3. Trace **nm([1, [3, 7], 5, 2)**
4. Trace **nm([1, [3, [2, 7], 4], 6])**

writing recursion with a scaffold

Now it's time for you to implement a function where we have provided a docstring. In order to help you we have chosen docstring examples to illustrate (a) a **base case**, where no recursive sub-calls are needed, and (b) a **general case** where you combine recursive sub-calls to return the desired result.

No tracing is required.

length

Create a subdirectory called **lab03**, and within it open file **length.py**. We'll define the **length** of a Python object **L** as:

1. 0 if **L** is not a list
2. maximum of **len(L)** and the **length** of any of **L**'s sublists, if **L** is a list

Read the docstring carefully, then fill in the implementation.

```
def length(L):
    ''' (list or non-list) -> int

    Return 0 if L is not a list, or the maximum of len(L)
           and the lengths of L's non-empty sublists.

    Assume: L is a non-list, or a non-empty list

    >>> length(17)
    0
    >>> length([1, [2, 3, 4, 5], 6])
    4
    >>> length([1, [2, 3, [4, 5]], 6])
    3
    '''
```

Show your work to your TA when you're done.

additional exercises

For each of these additional functions, trace the examples using the technique of this lab. At the end of the lab we will have a brief quiz resembling one of the examples in the lab or the additional exercises.