Lab 13: Recursion

Mar 20, 2018

Main Event

1. Setup and instructions

This lab comes with more infrastructure than others. Please take a moment to understand the framework.

Files

To begin, download the following files:

- helpers.py Contains three methods that you will need to complete this lab:
  1. `head` returns the head of a list.
  2. `tail` returns the tail of a list.
  3. `month_days` returns a list corresponding to days in each month of the year: index zero corresponds to the number of days in January, index one to February, and so on.
  4. `elves` returns a list of words that are “elfish”.

  The latter two methods are only useful if you decide to do the additional practice problems.
- lab.py This is where you will do your development. By default, the file contains function definitions that are correct, and return values that are incorrect. Your job is to replace the return values—and thus the function body’s—with correct implementations.
- test.py Contains methods that test your code. Do not edit this file. Instead, run this script once you feel you have completed a function in your lab file.

When run, the script evaluates each uncommented function in lab.py *in the order that is specified in the lab*. Once one of the functions fails an evaluation, the program stops.

All files should be in the same directory.

Rules

Remember, **this is a lab on recursion**! You cannot access list elements using subscript (bracket) notation, or using list methods. List elements can only be accessed using the functions in the helpers module:

```
>>> import helpers
>>> helpers.head([1, 2, 3])
1
>>> helpers.tail([1, 2, 3])
[2, 3]
```

Aside from these two functions, you can use native list concatenation and list creation where appropriate. Further, your code should not have any loops: no for-loops and no while-loops.

You may also be tempted to write “helper” methods that re-tool these problems—all of these methods can be done without it. Challenge yourself to think recursively!

2. Basics

1. Implement `rsum` such that, given a list of numbers, it returns the sum of that list. This is the exact problem we talked about in class; we’re doing it again just to get warmed up.
2. The function `exponentiate` should return `base` to the power of `power`:

   ```
   >>> 2 ** 10 == exponentiate(2, 10)
   True
   ```

   You cannot use `**`! Hint: exponentiation can be written as a recurrence relation.
3. We typically use subscript notation to return an element of a list. The function `get_nth` should do this without it:

```
>>> h = ['a', 'b', 'c']
>>> h[2] ==
'c'
>>> get_nth(h, 2)
'c'
```

4. Reverse a list (`reverse`) without using slicing or loops:

```
>>> reverse([1, 2, 3])
[3, 2, 1]
```

5. The function `is_older` takes two dates. It should return True if the first date (argument 1) comes before the second (argument 2); it should return False otherwise:

```
>>> is_older([2015, 10, 27], [2015, 10, 28])
True
```

   For this exercise (and subsequent additional practice exercises), we consider a “date” to a be a list of three integers. The first element is the year, the second element is the month, and the third element is the day.

Solution:

```
import helpers

def rsum(values):
    if not values:
        return 0
    else:
        return helpers.head(values) + rsum(helpers.tail(values))

def exponentiate(base, power):
    if power < 1:
        return 1
    else:
        return base * exponentiate(base, power - 1)

def get_nth(list_of, n):
    if n == 0:
        return helpers.head(list_of)
    else:
```

```
        return get_nth(helpers.tail(list_of), n - 1)

def reverse(list_of):
    if not list_of:
        return []
    else:
        return reverse(helpers.tail(list_of)) + [ helpers.head(list_of

def is_older(date_1, date_2):
    if not date_1 or not date_2:
        return False
    else:
        d1 = helpers.head(date_1)
        d2 = helpers.head(date_2)
        if d1 != d2:
            return d1 < d2
        else:
            return is_older(helpers.tail(date_1), helpers.tail(date_2)
```

Additional Practice

1. More basics

1. The function `number_before_reaching_sum` takes an integer, `total`, and a list of integers `numbers`. It should return an integer *n* such that the first *n* elements of the list sum to less-than `total`, but the first *n* + *1* elements of the list sum to *total* or more:

   ```
   >>> number_before_reaching_sum(6, [0, 1, 2, 3, 4])
   3
   >>> number_before_reaching_sum(7, [0, 1, 2, 3, 4])
   4
   ```

   You can assume that the entire list sums to more than the specified value, and that the specified value is positive.

2. The function `what_month` takes a day of the year—an integer between 1 and 365—and returns the month, as an integer, that that day is in:

   ```
   >>> what_month(180)
   6
   ```

The function `helpers.month_days` returns a list of integers corresponding to days in each month.

Solution:

```
def number_before_reaching_sum(total, numbers):
    total -= helpers.head(numbers)
    if total < 1:
        return 0
    else:
        return 1 + number_before_reaching_sum(total, helpers.tail(numb

def what_month(day):
    return number_before_reaching_sum(day, helpers.month_days()) + 1
```

2. Elfish words

Given two words, `elfish` should use recursion to decide whether all of the letters in the second word, in any order, are contained in the first. If so, it should return True; it should return False otherwise.

The method `helpers.elves` returns a list of words that all contain the letter e, l, and f, in some order (the words are thus cononically known as being “elfish”). Thus,

```
for i in helpers.elves():
    if elfish(i, 'elf'):
       print(i)
```

would print every word in the list.

Solution:

```
def elfish(word, elf='elf'):
    if not elf:
        return True
    elif not word:
        return False
    else:
        c = word[0]
        i = elf.find(c)
```

```
        elf_ = elf if i < 0 else elf[:i] + elf[i+1:]

        return elfish(word[1:], elf_)
```

3. Coin game

“Coin Game” consists of two players sitting in front of a mound of coins. Each turn consists of a player taking one, two, or four coins from the mound. The objective is to be the player that takes the last coin.

1. Given the number of coins in the mound, `coin_game_strategies` should return the number of strategies there are for the potential winner to win *if each player is playing optimally*.
2. Given the number of coins, along with the two players—as two distinct string arguments, in playing order— `coin_game_winner` should return the player (as a string) who will ultimately win.

Assume that both players are very smart and will try their best to work out a strategy to win the game. For example, assume Alice and Bob are the players. if there are two coins and Alice is the first player to pick, she will definitely pick two coins and win. If there are three coins and Alice is still the first player to pick, regardless of whether she pics one or two coins, Bob will get the last coin and win the game.

Solution:

```
# Solution inspired by cx473

def coin_game_strategies(coins):
    if coins < 0:
        return 0
    elif coins < 3:
        return 1
    elif coins == 3:
        return 2
    else:
        mask = [
            '111',
            '101',
            '010',
        ]
```

```
        tally = 0

        for (i, bit) in enumerate(mask[coins % 3]):
            if int(bit):
                tally += coin_game_strategies(coins - 2 ** i)

        return tally

def coin_game_winner(coins, player_1, player_2):
    return player_1 if coins % 3 else player_2
```

Introduction to Computer Science

Introduction to Computer Science
jerome.white@nyu.edu

jerome-white

Learning computer science concepts through practice.