

CS 1110 review session for Prelim 2, April 2018

SOLUTIONS

Questions chosen to cover a good range of topics, but this set of questions together would be probably too hard/long to be all on a single prelim. We will cover the few questions in class and leave the rest as at-home exercises.

1. [points] **Classes** (2014 SP) Classes `Course`, `Student`, and `Schedule` (see [handout](#)) are part of the Registrar's database, which records which courses each student is enrolled in, and which students are enrolled in each course. Two methods are unimplemented: `Student.add_course` (line 84), which updates the database to reflect a student enrolling in a course, and `Student.validate` (line 92), which makes sure a student's schedule follows the rules.

Read the code to become familiar with these classes. The 3rd page is a set of tests to help you to understand how these classes are used. Implement the two incomplete methods below.

```
class Student():
    ...
    def add_course(self, course):
        """Add a course for the current semester. This means the course
        is added to the student's current schedule, and the student is
        added to the enrollment of the course.
        Pre: <course> is a Course, the student has a current schedule, and
            <course> is not already on current semester's schedule."""
```

Solution:

```
self.schedules[0].courses.append(course)
course.students.append(self)

def validate(self, credit_limit):
    """Return: True if the student's schedule for the current semester
    is valid, which means that
    (a) the total number of credits in current semester is not over
        <credit_limit> (credits from prior semesters don't matter)
    (b) student is not taking any courses in current semester that
        they already took in a previous semester. Course titles
        determine when a course is repeated; see Schedule.overlaps.
    Pre: credit_limit [integer] ; student has a current schedule."""
```

Solution:

```
current_sem = self.schedules[0]
# check for overlaps, return False if any
for prev in self.schedules[1:]:
    if current_sem.overlaps(prev):
        return False

# Now we know there are no overlaps
return current_sem.total_credits() <= credit_limit
```

2. [points] **Iteration and recursion on a sequence. From 2013 Spring**

The motivation for this problem is analyzing patterns in a sequence. We would argue that this is a case where recursion is more natural and easier to implement than use of a for-loop.

Finish the implementation of the following function; note that we have started it off for you. Your implementation must use a loop to determine the length of the first run of zeroes in the list `data`, and then use recursion to determine the final answer.

```
def max_zrun_length(data):  
    """Returns: length of the longest run of zeroes in data, which is a  
    list of ints (possibly empty).  
  
    Examples: max_zrun_length([0,0,1,0,0,0]) is 3  
    max_zrun_length([2,0,4,0,5]) is 1  
    max_zrun_length([3,-2]) is 0  
    max_zrun_length([]) is 0  
    """  
    if 0 not in data:  
        | return 0  
  
    # now data is guaranteed to contain at least one 0  
    i = data.index(0) # i is the position of the first 0
```

Solution:

```
# Inv: data[i..j-1] is known to be 0, so j is the next place to check  
j = i + 1  
while j < len(data) and data[j] == 0:  
    j += 1  
  
firstlength = j - i # length of first run of zeroes.  
return max(firstlength, max_zrun_length(data[j:]))  
# data[j:] is empty if j == len(data)  
  
##### Alternate solution: #####  
# inv: firstlength is length of run seen so far;  
# i is index of last zero seen, so i+1 is next place to check  
firstlength = 1  
while i+1 < len(data) and data[i+1] == 0:  
    firstlength += 1  
    i=i+1  
return max(firstlength, max_zrun_length(data[i+1:]))
```

3. [points] **Recursion on nested lists. From Spring 2014 lecture.**

```
def embed(theinput):  
    """Returns: depth of embedding, or nesting, in theinput.
```

Examples:

```
    "the dog that barked" -> 0  
    ["the", "dog", "that", "barked" ] -> 1  
    ["the" ["dog", "that", "barked"]] -> 2  
    ["the" ["dog", ["that", "barked"]], ["was bad"]] -> 3  
    ["the" ["dog", ["that", ["barked"]]]] -> 4  
    [[["the"], "dog"], "that", "barked"] -> 4
```

```
Precondition: theinput is a string, or a potentially nested  
non-empty list of strings. No component list can be empty"""
```

Solution:

Solution using map. Recall that `map(f, alist)` returns the list `[f(alist[0]), f(alist[1]), ..., f(alist[len(alist)-1])]`.

```
    if type(theinput) != list: # no sub-lists to check  
        return 0  
    else:  
        return 1 + max(map(embed, theinput))
```

Solution using for-loop instead of map:

```
    if type(theinput) != list:  
        return 0  
  
    # sublists to account for  
    sublist_max = 0  
    for item in theinput:  
        sublist_max = max(sublist_max, embed(item))  
    return 1 + sublist_max
```

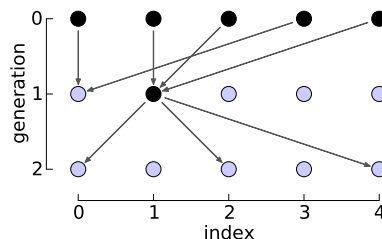
4. [points] **Recursion on object structures. Modified from 2014 spring.** Assume we have written a definition for class `Node` where each node has a `contacted_by` attribute consisting of a (possibly empty) list of nodes that have contacted it.¹ This question asks you to add a new method for class `Node`; implement it according to its specification. Your solution must make effective use of recursion, though it can involve for-loops as well.

```
class Node():
    ...
    def is_downstream_from(self, older):
        """Returns True if: older is in this node's contacted_by list, OR if
           at least one of the nodes in this node's contacted_by list is
           downstream from older. Returns False otherwise.
           Pre: older is a node.
        """
```

Solution:

```
if older in self.contacted_by:
    return True
else:
    # if self.contacted_by is [], the for-loop never happens
    for contactor in self.contacted_by:
        if contactor.is_downstream_from(older):
            return True
# If we get to this line, no contactor was downstream of older
return False
```

Example: In the figure below, (2,0) is downstream from (0,1), (0,2), (0,4), and (1,1), but no other nodes. (If you are looking at this after 2014 spring, note that you don't need to know what "generation" and "index" refer to in the figure.)



¹And we know that anything in a node's `contacted_by` list is from an earlier "generation". This is a technical condition that prevents the possibility of cycles in the node contacting.

5. [points] **Classes and Subclasses. Modified from Fall 2016 prelim 2.**

We've included a Fall prelim question on this handout so that you could see how Fall prelim questions of this sort would be modified for our semester.

You are to complete the skeleton of class `ExoticPet`, which is a subclass of `Pet`.

First, here are the specifications you need for class `Pet`; you don't need to know how its methods are implemented.

```
class Pet():
    """Instances represent a person's pet.
    MUTABLE ATTRIBUTES
        name: The name of the pet [nonempty string]
        tag: The license number for this pet [int >= -1]
    We use a tag value of -1 to indicate a pet without a license."""

    def __init__(self, name, tag = -1):
        """Initializer: Makes a pet have the given name and (optional) tag.
        Precondition: Parameter name is a nonempty string
        Precondition: Parameter tag is an int >= -1 (optional, default is
        -1)"""
        # Implementation omitted

    def __str__(self):
        """Returns: A string representation of this pet.
        String is the pet name with the license number in parentheses.
        A pet with a tag of -1 is "unclaimed".
        Examples: 'Sparky (pet #43)' or 'Rover (unclaimed)'''
        # Implementation omitted
```

What you have to do for the ExoticPet class definition:

1. Fill in the missing information in the class header.
2. Implement each method according to its specification.

```
class ExoticPet(          Pet          ):          # Fill in missing part
    """Instances represent a pet certified by a wild-life official.
    ATTRIBUTE (In addition to those from Pet):
        official: The official certifying the pet [nonempty string] """

    def __init__(self, name, tag, official):
        """Initializer: Makes an exotic pet have name <name>, tag <tag>, and
        official <official>.

        In the case of an exotic pet, the tag is not optional and must be a
        value greater than -1.

        Precondition: Parameter name is a nonempty string
        Precondition: Parameter tag is an int >= 0
        Precondition: Parameter official is a nonempty string"""
        Pet.__init__(self, name, tag)
        self.official = official

    def __str__(self):
        """Returns: A string representation of this exotic pet

        The string is the same as for Pet, except that it also includes the
        official who certified the pet.

        Example: 'Tigger (pet #675); authorized by Sgt. O'Malley'"""
        result = Pet.__str__(self)
        result += '; authorized by ' + self.official
        return result
```