

REVIEW POTPOURRI AND SQL

COMPUTER SCIENCE MENTORS

November 16, 2020 - November 19, 2020

1 SQL

CS 61A wants to start a fish hatchery, and we need your help to analyze the data we've collected for the fish populations! Running a hatchery is expensive – we'd like to make some money on the side by selling some seafood (only older fish of course) to make delicious sushi.

The table `fish` contains a subset of the data that has been collected. The SQL column names are listed in brackets.

Table name: `fish`*

Species [species]	Population [pop]	Breeding Rate [rate]	\$/piece [price]	# of pieces per fish [pieces]
Salmon	500	3.3	4	30
Eel	100	1.3	4	15
Yellowtail	700	2.0	3	30
Tuna	600	1.1	3	20

*(This was made with fake data, do not actually sell fish at these rates) The table `competitor` contains the competitor's price for each species.

Species [species]	\$/piece [price]
Salmon	2
Eel	3.4
Yellowtail	3.2
Tuna	2.6

1. Business is good, but a bunch of competition has sprung up! Through some cunning corporate espionage, we have determined one such competitor's selling prices.

Write a query that returns, for each species, the difference between our hatchery's revenue versus the competitor's revenue for one whole fish.

For the following two questions, you have access to two tables.

Grades, which contains three columns: `day`, `class`, and `score`. Each row represents the `score` you got on a midterm for some `class` that you took on some `day`.

Outfits, which contains two columns: `day` and `color`. Each row represents the `color` of the shirt you wore on some `day`. Assume you have a row for each possible `day`.

Table name: `grades`

Day	Class	Score
10/31	Music 70	88
9/20	Math 1A	72

Table name: `outfits`

Day	Color
11/5	Blue
9/13	Red
10/31	Orange

1. Instead of actually studying for your finals, you decide it would be the best use of your time to determine what your "lucky shirt" is. Suppose you're pretty happy with your exam scores this semester, so you define your lucky shirt as the shirt you wore to the most exams.

Write a query that will output the color of your lucky shirt and how many times you wore it.

2 Scheme

1. Fill in `skip-list`, which takes in a potentially nested list `lst` and a single-argument filter function `filter-fn` that returns a boolean when called, and goes through each element in order. It returns a new list that contains all elements that return true when passed into `filter-fn`. The returned list is *not nested*.

```
;Doctests
```

```
scm> (skip-list '(1 (3)) even?)
```

```
()
```

```
scm> (skip-list '(1 (2 (3 4) 5) 6 (7) 8 9) odd?)
```

```
(1 3 5 7 9)
```

```
(define (skip-list lst filter-fn)
  (define (helper lst lst-so-far next)
    (cond
      ((null? lst)
       (if (null? _____)
           _____
           _____)
       )
      ((pair? _____)
       (_____))
      (else
       _____)
    )
  )
  (helper _____))
```

3 Environment Diagrams

1. Draw the environment diagram that results from running the following code.

```
ghost = [1, 0, [3], 1]
def boo(spooky):
    ghost.append(spooky.append(ghost))
    spooky = spooky[ghost[2][1][1]]
    ghost[:].extend([spooky])
    spooky = [spooky] + [ghost[spooky - 1].pop()]
    ghost.remove(ghost.remove(1))
    spooky += ["Happy Halloween!"]
    return spooky
pumpkin = boo(ghost[2])
```

4 Recursion

1. Suppose the **position** of the rightmost digit of a number is defined as 1. The position of a digit increases from right to left.

(a) Fill in the `positionizer` function below. It takes in a non-negative integer `n` and returns a non-negative integer. For each digit `d` of `n`, `positionizer` either changes `d` to be the remainder of `d` divided by its position, or leaves `d` the same if it is equal to its position.

```
def positionizer(n):  
    """  
    >>> positionizer(12)  
    10  
    >>> positionizer(23)  
    20  
    >>> positionizer(12345)  
    12300  
    """  
    def helper(n, pos):  
  
        if _____:  
  
            return _____  
  
        rest = _____  
        if n % 10 == pos:  
  
            return rest + _____  
        else:  
  
            return rest + _____  
  
    return helper(_____, _____)
```

(b) Now fill in the `max_positionizer` function below. Given a list of digits, it returns the k -digit number with the highest value when positioned. The digits must stay in the same order. (ex: for $k = 2$ and `lst = [1, 2, 3]`, you can't form 21). You may use `positionizer` in your solution.

Hint: Use the helper to create a list of all possible k digit numbers formed from `lst`.

```
def max_positionizer(k, lst):
    """
    >>> max_positionizer(2, [1, 2, 3]) # positionized version
                                           # of 12, 13, 23 are
                                           # 10, 10, 20

    23
    >>> max_positionizer(3, [2, 5, 3, 1])
    251
    """
    def make_nums(k, lst):

        if _____:

            return _____

        elif _____:
            return []

        a = [_____ \
            _____
            for rest in _____]

        b = _____
        return a + b

    return _____(make_nums(_____, _____), _____)
```

1. What would Python display? The questions continue on the next page.

```
class Food:
    def __init__(self, name, spoiled = False):
        self.name = name
        self.num_days = 0
        self.spoiled = spoiled

    def can_eat(self):
        self.num_days += 1
        if self.num_days >= 3:
            self.spoiled = True
            print("Oh no! Your food is spoiled!")
        return not self.spoiled

    def mix_food(self, other_food):
        self.num_days = self.num_days + other_food.num_days
        self.name += " " + other_food.name
        self.spoiled = self.spoiled and other_food.spoiled

class Salad(Food):
    def __init__(self, ingredients):
        super().__init__("salad", False)
        self.ingredients = ingredients

    def add_ingredients(self, ingredient):
        self.ingredients.append(ingredient)
        print(ingredient.name + " has been added")

    def mix_ingredients(self):
        for ingredient in self.ingredients:
            self.mix_food(ingredient)
        print("Your salad has been mixed.")

lettuce = Food("lettuce")
tomatoes = Food("tomatoes")
chicken = Food("chicken")
ingredients = [lettuce, tomatoes]
my_salad = Salad(ingredients)
```



```
>>> lettuce.can_eat ()  
  
>>> my_salad.can_eat ()  
  
>>> my_salad.mix_ingredients ()  
  
>>> my_salad.name
```

6 Trees

1. The total weight of a tree is defined as the sum of the labels of all its nodes. A tree is defined to be `equally_weighted` if the total weight of each of its branches are equal. A leaf is assumed to be equally weighted.

Complete the following functions for `equally_weighted` and `num_eq_weight`. You may use the below definition of the `total_weight` function in your answer.

```
def total_weight(t):
    """
    Return the total weight of a tree, i.e. the sum of all its
    labels.
    >>>total_weight(Tree(1, [Tree(2), Tree(3,[Tree(4)])]))
    10
    """
    weight = t.label + sum([total_weight(branch) for branch in
        t.branches])
    return weight
```

(a) **def** `equally_weighted(t)`:

```
    """
    Return whether a tree is equally weighted.
    >>>equally_weighted(Tree(1))
    True
    >>>equally_weighted(Tree(1,[Tree(2), Tree(1, [Tree(1)])]))
    True
    >>>equally_weighted(Tree(0, [Tree(3), Tree(2, [Tree(3)])]))
    False
    """
    _____ = [_____]

    for _____ in _____:

        if _____:

            return _____

    return _____
```

(b) **Note:** You are allowed to use `equally_weighted` in this part.

```
def num_eq_weight(t):
    """
    Return the number of equally weighted subtrees of t. Note
    that t is considered a subtree of itself.
    >>> num_eq_weight(Tree(1, [Tree(4), Tree(3, [Tree(1)])]))
    4
    >>> num_eq_weight(Tree(1, [Tree(9),
                               Tree(1, [Tree(4),
                                         Tree(3, [Tree(1)])])])
    )
    6
    >>> num_eq_weight(Tree(1, [Tree(8, [Tree(1)]),
                               Tree(1, [Tree(4),
                                         Tree(3, [Tree(1)])])])
    )
    7
    """
    val = _____

    if _____:

        return _____
    else:
        return val
```

7 Generators

1. Define a **non-decreasing path** as a path from the root where each node's label is greater than or equal to the previous node along the path. A **subpath** is a path between nodes X and Y , where Y must be a descendent of X (ex: Y is a branch of a branch of X).

- (a) Write a generator function `root_to_leaf` that takes in a tree `t` and yields all non-decreasing paths from the root to a leaf node, in any order. Assume that `t` has at least one node.

```
def root_to_leaf(t):
    """
    >>> t1 = Tree(3, [Tree(5), Tree(4)])
    >>> list(root_to_leaf(t1))
    [[3, 5], [3, 4]]
    >>> t2 = Tree(5, [Tree(2, [Tree(7), Tree(8)]), Tree(5,
        [Tree(6)])])
    [[5, 5, 6]]
    """

    if _____:

        _____

    for _____:

        if _____:

            for _____:

                _____
```

- (b) Write a generator function `subpaths` that takes in a tree `t` and yields all non-decreasing subpaths that end with a leaf node, in any order. You may use the `root_to_leaf` function above, and assume again that `t` has at least one node.

```
def subpaths(t):

    yield from _____

    for b in t.branches:

        _____
```

