UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Midterm Review

Announcements & Policies

- Midterm:
  - 2 hours, 120 Minutes
  - Unlimited **Handwritten** Cheat sheets – More than ~3 is counter-productive
  - 1 CS88 Provided Reference Sheet
  - Verify Scheduling / Accommodations

My Advice

- Don't rush!
- Skim the exam first
  - It's ok to do questions out of order!
  - Get the stuff you're good without out of the way
  - BUT don't spend too much time planning the exam.
- Read through the question once
  - What's it asking you to do at a high level?
  - What do the doctests suggest?
  - What techniques should you be using?

UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Questions from Ed

OOP Questions

- super() is a method which refers to the *parent class* of an object.
  - It returns a special kind of instance of an object.
  - We can use super() or the class name of the parent class
- Private Attributes and Methods:
  - self.x is "public"
  - self._x is a *convention* which means "you shouldn't use this"
  - self.__x is *private*. Python prevents other classes from accessing that attribute.
- Method calls:
  - object.method(arguments) is the same as Class.method(object, arguments)
  - cs88.deposit(50) vs Account.deposit(cs88, 50)

ADTs

- Constructors / Selectors / Operators
  - help us think about the actions to represent an object.
- Abstraction Barrier:
  - Means that a function does not rely *directly* on a particular implementation
- Should a function return data?
  - Usually yes! It depends on the goals!

Exam Practice

- Spring 22 Q7
- Spring 20 Q5

SP22

7. (5.0 points) Closet Overhaul

You've designed a closet abstract data type to help you organize your wardrobe.

A closet contains two things:

- `owner`: the name of the closet owner represented as a string
- `clothes`: the collection of clothes in the closet represented as a dictionary, where the key is the clothing item name and the value is the number of times the clothing item has been worn.

The `make_closet` constructor takes in `owner` (a string) and `clothes` (a **list** of strings representing clothing items) and returns a closet ADT.

Given this, you've implemented the abstract data type as follows:

```
def make_closet(owner, clothes):
    """ Create and returns a new closet. """
    clothes_dict = {}
    for item in clothes:
        clothes_dict[item] = 0
    return (owner, clothes_dict)

def get_owner(closet):
    """ Returns the owner of the closet """
    return closet[0]

def get_clothes(closet):
    """ Returns a dictionary of the clothes in the closet """
    return closet[1]
```

Given the closet ADT, implement the functions `wear_clothes` and `favorite_clothing_item`. You may not need all the lines provided, and you may need to change the indentation for some lines.

5. (10 points) Atey Ate Already

It's a lot more fun to think about food than take midterms, so let's look at the cheapest places to fulfill an order. Given the function `total_cost` and assuming it works as described, fill out `find_restaurant` to find the cheapest restaurant to fulfill the order.

Remember: Pay close attention to the doctests to guide your solution.

```
def total_cost(restaurant, order):
        """
        Function that returns the total cost of an order at a certain
        restaurant. Returns -1 if fulfilling the request is not possible.
        >>> total_cost('chipotle', ['burrito', 'taco'])
        11.96
        >>> total_cost('sliver', ['boba'])
        -1.0
        """
        # We have omitted how this function works.
def find_restaurant(restaurants, order):
    """
    Function that returns the cheapest restaurant and price as the first
    element of a list followed by the prices for each of the restaurants.
    In the case that no restaurant can fulfill the order, the first
    element should be ['None found!', -1]. In the case that two
    restaurants have the same price, keep the first restaurant.
    Hint: Use total_cost!
    >>> find_restaurant(['chipotle', 'la burrita'], ['burrito', 'taco'])
    [['la burrita', 9.78], [['chipotle', 11.96], ['la burrita',9.78]]

    >>> find_restaurant(['sliver','cheeseboard'], ['boba'])
    [[None found!, -1.0], ['sliver', -1.0]['cheeseboard', -1.0]]
    """
```

SP20 #6

(10 points) Rooms within Rooms within Rooms

You are a Data Scientist hired by UC Berkeley to find the largest room on campus. In order to schedule midterms, your job is return the room and its capacity. The data on all the rooms plus capacity is in a weird format of three element lists, where the first element is the room, the second element is the capacity, and the third element is either the rest of the data or None. Assume that the capacity of each of the rooms is unique.

That is, the data look like `['Room', Number, [...]]`.

Use the following lines of code to fill in the body of the function. You will need to fill in the blanks of the lines provided. Some lines are optional.

```
    return [rooms[0], rooms[1]]
    largest_left = find_largest(______________________)
    if rooms[2] == _______________:
    if largest_left[1] > rooms[1]:
    return ____________________
    return ____________________
    else: # this line is optional, depending upon your solution
    else: # this line is optional, depending upon your solution
```

```
def find_largest(rooms):
    """
    Return the largest room from a weirdly nested list.
    You can assume rooms is always 3 items long.
    >>> rooms = ['Evans', 150, ['Wheeler', 700, ['Stadium', 50000, None]]]
    >>> find_largest(rooms)
    ['Stadium', 50000]
    >>> find_largest(['Evans', 150, None])
    ['Evans', 150]
    """
```

```
def find_largest(rooms):
    """
    Return the largest room from a weirdly nested list.
    You can assume rooms is always 3 items long.
    >>> rooms = ['Evans', 150, ['Soda', 200, ['Wheeler', 700, ['Stadium', 50000, None]
    >>> find_largest(rooms)
    ['Stadium', 50000]
    >>> find_largest(['Evans', 150, ['Hearst Annex', 50, None]])
    ['Evans', 150]
    """
    if rooms[2] == None:
        return [rooms[0], rooms[1]]
    largest_left = find_largest(rooms[2])
    if largest_left[1] > rooms[1]:
        return largest_left
    return [rooms[0], rooms[1]]
```

Fall 2019 #6

6. (8 points) Time To Get Your Game On!

Long before NBA 2K came out, there was a much more primitive version of the video game, called NBA 1K. They stored NBA players' information in lists, and people who played the game would draft a team of these players. A team's score would be the sum of all individual player scores, and a player score would be the sum of points, rebounds, and assists multiplied by the `team_skill`® factor of the team they play for.

Fill in the functions below to calculate a team's score. Most of these functions can be written in 1 line. You will get credit for using a function correctly even if your implementation is not complete.

```
# Use the following code as a guide in your functions.
# You will not use any of these variables directly.
teams = [ [team1_name, team1_skill], [team2_name, team2_skill], ]
teams = [ ['Golden State', 2.0], ['Los Angeles Lakers', 1.5] ]

player = [name, team_name, points, rebounds, assists]
players = [
    ['Stephen Curry', 'Golden State', 40, 10, 20],
    ['LeBron James', 'Los Angeles Lakers', 20, 10, 5]
]


def NBA1k_score(teams):
```

Fa21 8c

i. **(4.0 pt)** Implement the `find_new_interest` method that returns a string representing a new potential interest for this `User`. To determine this new interest, first identify this user's most similar follower that has the largest number of mutual interests with this user. Then return a randomly selected interest from this follower. But be careful, this randomly selected interest must not already exist in this user's interests (otherwise it would not be new!).

For this problem, assume that the user's interests and followers are non-empty. Note that the `separate_interests` function (see `User` class skeleton) may be helpful here. You may use `random.choice(lst` to return a radomly selected item from a list, `lst`.

```
def find_new_interest(self):
    """
    >>> u1 = User('bob', ['cooking', 'archery', 'tv'])
    >>> u2 = User('alice', ['shopping', 'guitar', 'cooking']) # has one in common with bob
    >>> u3 = User('mike', ['poker', 'tv', 'cooking']) # has two in common with bob
    >>> u1.add_follower(u2)
    >>> u1.add_follower(u3)
    >>> u1.find_new_interest()
    'poker'
    """
    most_similar_follower = max(
        ____________________________________,
        key = ____________________________________
    )
    return random.choice(_____________________________________________)
```

Write the fully *completed* `find_new_interest` function below using the skeleton code provided. You may not add, change, or delete lines from the skeleton code.

SP20 #5

(10 points) Atey Ate Already

It's a lot more fun to think about food than take midterms, so let's look at the cheapest places to fulfill an order. Given the function `total_cost` and assuming it works as described, fill out `find_restaurant` to find the cheapest restaurant to fulfill the order.

Remember: Pay close attention to the doctests to guide your solution.

```
def total_cost(restaurant, order):
        """
        Function that returns the total cost of an order at a certain
        restaurant. Returns -1 if fulfilling the request is not possible.
        >>> total_cost('chipotle', ['burrito', 'taco'])
        11.96
        >>> total_cost('sliver', ['boba'])
        -1.0
        """
        # We have omitted how this function works.
def find_restaurant(restaurants, order):
    """
    Function that returns the cheapest restaurant and price as the first
    element of a list followed by the prices for each of the restaurants.
    In the case that no restaurant can fulfill the order, the first
    element should be ['None found!', -1]. In the case that two
    restaurants have the same price, keep the first restaurant.
    Hint: Use total_cost!
    >>> find_restaurant(['chipotle', 'la burrita'], ['burrito', 'taco'])
    [['la burrita', 9.78], [['chipotle', 11.96], ['la burrita',9.78]]

    >>> find_restaurant(['sliver','cheeseboard'], ['boba'])
    [[None found!, -1.0], ['sliver', -1.0]['cheeseboard', -1.0]]
    """
```

```
def findRestaurant(restaurants, order):
    """
    Function that returns the cheapest restaurant and price as the first
    element of a list followed by the prices for each of the restaurants.
    In the case that no restaurant can fulfill the order, the first
    element should be ['None found!', -1].
    Hint: Use totalCost! In the case that two restaurants have the same price,
    keep the first restaurant.
    >>> findRestaurant(['chipotle', 'la burrita'], ['burrito', 'taco'])
    [['la burrita', 9.78], [['chipotle', 11.96], ['la burrita',9.78]]
    >>> findRestaurant(['sliver','cheeseboard'], ['boba'])
    [[None found!, -1.0], ['sliver', -1.0]['cheeseboard', -1.0]]
    """
    placesList = [ [restaurant, totalCost(restaurant, order)]
                   for restaurant in restaurants ]
    minCost = -1.0
    cheapestPlace = "None found!"
    for place in range(placesList):
        if place[1] != -1.0 and (place[1] < minCost or minCost == -1.0):
            minCost = place[1]
            cheapestPlace = place[0]
    return [cheapestPlace, minCost] + placesList
```