CS 88
Fall 2019

Computational Structures in Data Science

Midterm Solutions

INSTRUCTIONS

- You have 2 hours to complete the exam. **Do NOT open the exam until you are instructed to do so!**
- The exam is closed book, closed notes, closed computer, closed calculator, except one hand-written 8.5" × 11" crib sheet of your own creation and the official CS 88 midterm 1 study guide.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper.

Full Name	
Student ID Number	
Official Berkeley Email (@berkeley.edu)	
TA	
Name of the person to your left	
Name of the person to your right	
By my signature, I certify that all the work on this exam is my own, and I will not discuss it with anyone until exam session is over. **(please sign)**	

POLICIES & CLARIFICATIONS

- If you need to use the restroom, bring your phone and exam to the front of the room.
- You may use built-in Python functions that do not require import, such as `min`, `max`, `pow`, `len`, and `abs`.
- You **may not** use example functions defined on your study guide unless a problem clearly states you can.
- For fill-in-the-blank coding problems, we will only grade work written in the provided blanks. You may only write one Python statement per blank line, and it must be indented to the level that the blank is indented.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.

1. (12 points) Evaluators Gotta Evaluate...

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. If an error occurs, write "Error", but include all output displayed before the error. If evaluation would run forever, write "Forever". To display a function value, write "Function". The first two rows have been provided as examples.

The interactive interpreter displays the value of a successfully evaluated expression, unless it is `None`.

Assume that you have first started `python3` and executed the statements on the left.

```
def example():
    print('Do Nothing')
    return None

x = 8
y = 88
z = 'python'

def w(d):
    x = 0
    while x > 0:
        return None
    return d + 1

def fun(f, x, y):
    f = min
    return f(x,y)
```

	Expression	Interactive Output
	`x * y`	704
	`example()`	Do Nothing None
(2 pt)	`x or y`	8
(2 pt)	`(x and y) * (x/y)`	8.0
(2 pt)	`str(x) == y / x - 3`	False
(2 pt)	`'Py' + z`	'Pypython'
(2 pt)	`fun(max, 61, 88)`	88

(2 pt)	`lam = lambda x: lambda y: w(x)`
	`lam(2)(3)`
		3

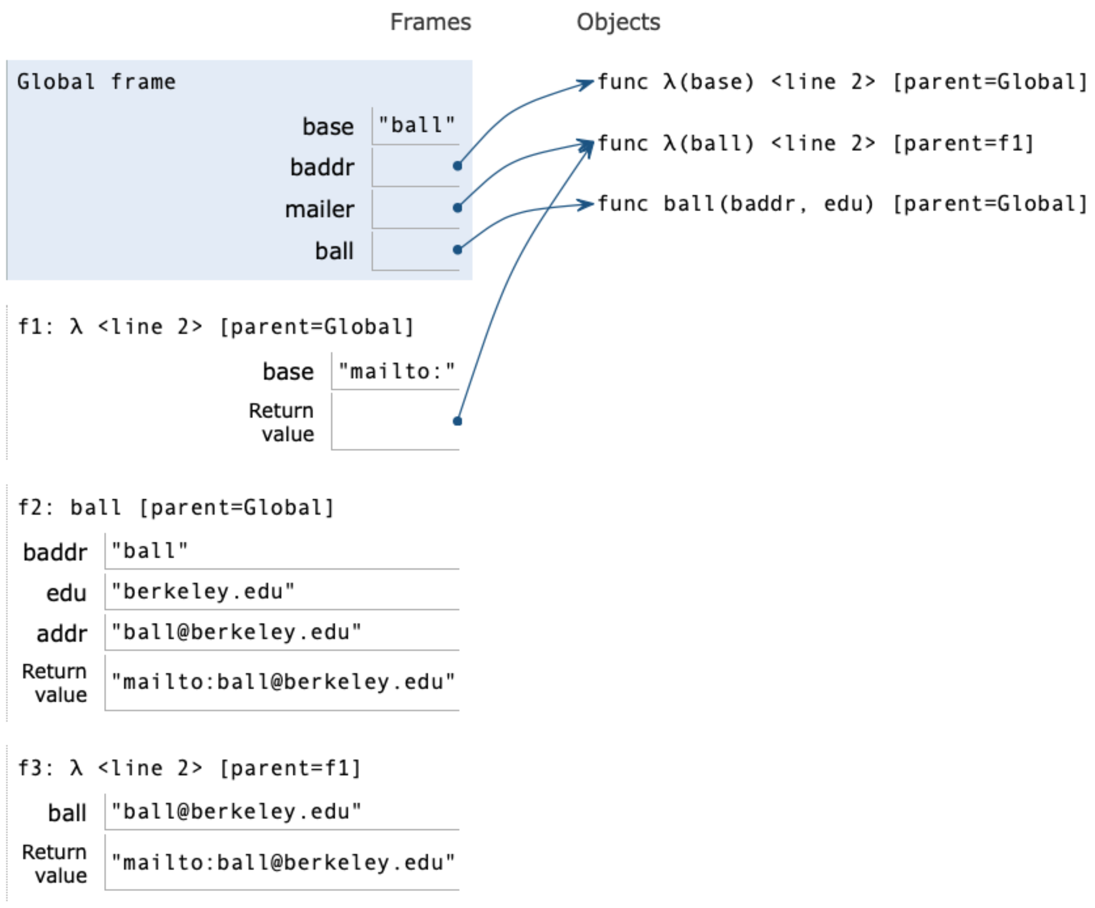
2. (8 points) Save the Environment!

Fill in the environment diagram that results from executing the code on the right until the entire program is finished, an error occurs, or all frames are filled. We have started the environment diagram for you.

A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.

```
base = "ball"
baddr = lambda base : lambda ball: base + ball
mailer = baddr("mailto:")
def ball(baddr, edu):
  addr = baddr + "@" + edu
  print(baddr + " got mail")
  return mailer(addr)
ball(base, "berkeley.edu")
```



Frames
Objects
Global frame
base "ball"
baddr
mailer
ball
func λ(base) <line 2> [parent=Global]
func λ(ball) <line 2> [parent=f1]
func ball(baddr, edu) [parent=Global]
f1: λ <line 2> [parent=Global]
base "mailto:"
Return value
f2: ball [parent=Global]
baddr "ball"
edu "berkeley.edu"
addr "ball@berkeley.edu"
Return value "mailto:ball@berkeley.edu"
f3: λ <line 2> [parent=f1]
ball "ball@berkeley.edu"
Return value "mailto:ball@berkeley.edu"

3. (10 points) Happy, Happy Halloween!

It's almost Halloween! As college students, we know that trick-or-treating is all about maximizing the value of our candy haul by only filling our bags to capacity with the most valuable candy.

Fill in the function `trick_or_treat` that will return a list of the names of the candies that maximizes the value of the candy we collect in our bag. For this problem, the 'value' of a candy is the length of the name of the candy. We only put one of each candy in our bag, and we cannot fill our bag beyond its capacity. (A bag with a value of 9 would be full if it contained a "twix" (4) and "M&M's" (5)). We can have any number of candies, but we can't go beyond the maximum value. (You can think of this like a bag that has maximum weight it can hold before breaking.) **We will maximize our haul by always taking the most valuable candy first.**

You may use the Python `max` function, which takes in a list and returns the largest value in the list. You may also use the Python remove function, which removes an element from a list with the following syntax: `list.remove(element)`.

```
def trick_or_treat(candies, cap):
    """
    candies: a list of strings with the names of the candies
    cap: a number that is the capacity of the bag

    >>> trick_or_treat(['snickers', 'twix'], 5)
    ['twix']
    >>> trick_or_treat(['kit kat', 'dots', 'reeses'], 14)
    ['kit kat', 'reeses']
    """
    total = 0 # track value in our bag
    seen = 0 # count candies we have checked
    result_bag = [] # our result
    copy_candies = candies.copy() # don't modify our input
     # stop when we fill our bag, or check all candy
    while cap > total and seen < len(candies):
        lengths = [len(x) for x in copy_candies] # Get values
        curr_best = max(lengths) # find the largest value
        for candy in copy_candies:
            if len(candy) == curr_best: # find the candy that's the largest
                if total + len(candy) <= cap: # See if it fits in the bag
                    total += len(candy)
                    result_bag += [candy]
                copy_candies.remove(candy) # make sure we don't double-count.
            seen += 1
    return result_bag


def trick_or_treat_2(candies, cap):
    """An alternative solution... (condensed a bit)"""
    total, result_bag = 0, []
    copy_candies = candies.copy()
    while cap > total and len(copy_candies) > 0:
        lengths = [len(x) for x in copy_candies]
        curr_best = max(lengths)
        # `index` returns the index or position of that item.
        candy = copy_candies.pop(lengths.index(curr_best))
        if total + curr_best <= cap:
            total += curr_best
            result_bag += [candy]
    return result_bag
```

4. (8 points) Don't Tell A Fib!

Remember the Fibonacci sequence? Great, now let's change it up. Implement a recursive function that takes in a number n, and returns the nth element of a sequence where elements at an even index are the sum of the previous two elements, and elements at an odd index are the product of the previous two elements. The first two elements of the sequence are 0 and 1.

The sequence looks something like this:

0, 1, 1, 1, 2, 2, 4, 8, 12, 96, 108...

For example: $alt\_fib(9) = 96 = 12 * 8$ and $(alt\_fib(10) = 108 = 98 + 12$.

```
def alt_fib(n, counter=1):
    """
    >>> alt_fib(10)
    108
    >>> alt_fib(0)
    0
    >>> alt_fib(1)
    1
    """
    if n < 2:
        return n
    elif n % 2 == 0:
        return alt_fib(n - 1) + alt_fib(n - 2)
    else:
        return alt_fib(n - 1) * alt_fib(n - 2)
```

Count the calls: How many calls of `alt_fib` are made when we execute `alt_fib(6)`? (The initial call of `alt_fib(6)` counts as 1 call.)

______________________ 26 calls

Extra Credit (1 point): What happens when you call `alt_fib(-2)`? Use the code from your solution. (There are multiple possible correct answers, depending upon your previous answer.) *This is extra credit; come back to this only if you have time!*

If you used $n < 2$ as your base case, then the function should have returned -2 and would stop executing right then.

If you used $n == 0$ as a base case, then you likely had an infinite loop.

5. (20 points) Dinner Time!

It's dinner time! We're writing a function to check if we're having a complete meal.

Unscramble the following lines to complete the body of `isCompleteMeal`. You will use each line exactly once.

Given a list of 3 elements containing food items, return *True* if the 3 food items make a complete meal, meaning that it has one item from each category of appetizer, main, and dessert in no particular order. Return *False* if the meal is incomplete.

```
has_app _______________
has_app _______________
return _______________ and _______________ and _______________
has_dessert _______________
has_dessert _______________
elif _______________:
elif _______________:
for _______________:
if _______________:
has_main _______________
has_main _______________

def isCompleteMeal(apps, mains, desserts, meal):
        """
        Given a list of 3 elements containing food items,
        return True if the 3 food items make a complete meal,
        meaning that it has one item from each category of appetizer,
        main, and dessert in no particular order. Return False if the meal
        is incomplete.

        >>> appetizer = ["salad", "bread", "soup"]
        >>> main = ["pasta", "noodles", "steak"]
        >>> dessert = ["cake", "pie", "ice cream"]
        >>> meal = ["bread", "salad", "ice cream"]
        >>> isCompleteMeal(appetizer, main, dessert, meal)
        False
        >>> meal = ["bread", "steak", "ice cream"]
        >>> isCompleteMeal(appetizer, main, dessert, meal)
        True
        """
        app_count = 0
        main_count = 0
        des_count = 0
        for item in meal:
                if item in apps:
                        app_count += 1
                elif item in mains:
                        main_count += 1
                elif item in desserts:
                        des_count += 1
        return app_count == 1 and main_count == 1 and des_count == 1

def isCompleteMeal(apps, mains, desserts, meal):
        has_app, has_main, has_dessert = False, False, False
        for item in meal:
                if item in apps:
```

```
                has_app = True
            elif item in mains:
                has_main = True
            elif item in desserts:
                has_dessert = True
        return has_app and has_main and has_dessert
```

Use your `isCompleteMeal` function to fill in `numCompleteMeals`. You can get full points for this part even if your previous function isn't perfect. However, in order to get full credit, your solution must fit in one line of Python.

If you can't figure out a one line solution, but have a correct solution, you'll still get some points! Oh, and if you write big, don't worry about that. "One line" means one return statement.

```
def numCompleteMeals(app, main, des, food):
        """
        Given a list of 3 element lists containing food items,
        return the number of lists that make a complete meal,
        meaning that the list has one item from each category of appetizer,
        main, and dessert.

        >>> appetizer = ["salad", "bread", "soup"]
        >>> main = ["pasta", "noodles", "steak"]
        >>> dessert = ["cake", "pie", "ice cream"]
        >>> food = [["bread", "salad", "ice cream"], ["pasta", "pie",
        "salad"],["steak", "soup", "cake"], ["cake", "pie", "ice cream"]]
        >>> numCompleteMeals(appetizer, main, dessert, food)
        2
        """
        return sum([1 for meal in food if isCompleteMeal(app, main, des, meal)])
```

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):
    def get_team_skill(team_name):
        """
        >>> teams = [['Golden State', 2.0], ['Los Angeles Lakers', 1.5]]
        >>> get_team_skill('Golden State')
        2.0
        """
        return [team[1] for team in teams if team[0] == team_name][0]


    def calculate_player_score(player):
        """
        >>> calculate_player_score(['Stephen Curry', 'Golden State', 40, 10, 20])
        140
        """
        points, rebounds, assists = player[2], player[3], player[4]
        return (points + rebounds + assists) * get_team_skill(player[1])

    return calculate_player_score


def calculate_teams_score(teams, players):
    """
    >>> players = [['Stephen Curry', 'Golden State', 40, 10, 20],
        ['LeBron James', 'Los Angeles Lakers', 20, 10, 10]]
    >>> teams = [['Golden State', 2.0], ['Los Angeles Lakers', 1.5]]
    >>> calculate_teams_score(teams, players)
    200
    # Steph gets (40 + 10 + 20) * 2 points and
    # LeBron gets (20 + 10 + 10) * 1.5 points = 140 + 60 = 200
    """
    player_score_calculator = NBA1k_score(teams)
    return sum([player_score_calculator(player) for player in players])
```

7. (8 points) Lists, and Lists, and Lists, Oh My!

Given a list of lists of variable length that contains numbers, have `flatten_list` return a list of just those numbers maintaining the same order that they were in. Fill in your solution below. You may not need all lines.

```
def flatten_list(lst):
    """
    >>> flatten_list([[1, 2, 3], [4, 5], [6, 7, 8], [9]])
    [1, 2, 3, 4, 5, 6, 7, 8, 9]
    """
    result = []
    for item in lst:
        for num in item:
            result.append(item)
    return result

def flatten_list_recursive(lst):
    """
    This is an example of a recursive solution, you didn't need to solve
    this, but here's how you might.
    >>> flatten_list_recursive([[1, 2, 3], [4, 5], [[6, 7, 8]], [[[[9]]]]])
    [1, 2, 3, 4, 5, 6, 7, 8, 9]
    """
    result = []
    def flattener(lst, result):
        if not isinstance(lst, list):
            result.append(lst)
        elif len(lst) == 0:
            return
        else:
            flattener(lst[0], result)
            flattener(lst[1:], result)
    flattener(lst, result)
    return result
```

Extra Credit (2 points): Our function so far only works with 2-Dimensional lists (one level of nesting). Describe in two sentences (or so) how you would adapt this to handle an infinite level of nesting. *This is extra credit; come back to this only if you have time!*

____

____

____

In order to handle infinitely nested lists, we would need to make our solution *recursive.* (1 pt for this). To do this, we'd rewrite our function so that we call a helper function that recurses whenever it sees a new list inside our input list. (1 point for mentioning anything more about how you might adapt your solution. There are many possible answers.)

We have provided an example recursive solution, for you own interest. You did not need to write one.