DATA C88C Computational Structures in Data Science
Spring 2025

Midterm

INSTRUCTIONS

- **Do NOT open the exam until you are instructed to do so!**
- You have 120 minutes to complete the exam (unless you have an extended time accommodation).
- The exam is closed book, closed notes, closed computer, closed calculator, except for a cheatsheet of your own creation that follows the course guidelines and the provided reference sheet.
- Mark your answers on the exam itself in the spaces provided. We will not grade answers written on scratch paper or outside the designated answer spaces.
- Multiple choice questions are indicated by circular bubbles. You should select **one** option and **fill in the entire bubble**.
- Multi-select questions are indicated by square boxes. You should choose **one or more** options and **fill in the entire box**.
- If you need to use the restroom, bring your phone, student ID, and exam to the front of the room.
- You may use built-in Python functions that do not require import, such as `pow`, `len`, `abs`, `bool`, `int`, `float`, `str`, `round`, `max`, and `min`.
- You **may not** use example functions defined on your reference sheet unless a problem clearly states you can.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.
- You may not use `;` to place two statements on the same line.

Full Name	
Student ID Number	
Email (@berkeley.edu)	
What room are you in?	
Student ID of the person to your left	
Student ID 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)**	

- **Please write your SID at the top of each page!**
- **You must include all answers within the boxes.**
- If you must write outside the box, please draw an arrow.
- Use the blank space as scratch paper to work out your solutions.

1. (7.0 points) What Would Python Do?

The `Flower` class represents a flower with attributes for its name, number of petals, and color. The name of a flower is unique.

```
class Flower:
    def __init__(self, name, num_petals, color):
        self.name = name
        self.num_petals = num_petals
        self.color = color
        self.bee_visits = []

    def describe(self):
        print(f"The {self.name} is {self.color} and has {self.num_petals} petals.")

    def add_visit(self, bee_name):
        if bee_name not in self.bee_visits:
            self.bee_visits.append(bee_name)
```

For each of these questions, assume we have correctly created two `Flower` instances, `rose` and `daisy`, with the following attributes:

```
rose = Flower("rose", 30, "blue")
daisy = Flower("daisy", 56, "yellow")
```

(a) (1.0 pt) Select the output displayed by the interactive Python Interpreter when the code below is evaluated. The output may have multiple lines.

```
description = rose.describe()
print(description)
```

- ● Line 1: `The rose is blue and has 30 petals.` Line 2: `None`
- ○ Line 1: `The rose is blue and has 30 petals.`
- ○ Line 1: `The rose is blue and has 30 petals.` Line 2: `The rose is blue and has 30 petals.`

(b) (1.0 pt) What is the result of the following lines of code?

```
daisy.add_visit("small bee")
rose.add_visit("small bee")
daisy.bee_visits is rose.bee_visits
```

- ○ True
- ● False
- ○ The code leads to an error.

(c) **(1.0 pt)** Select all statements that will evaluate to `True` after the code below is executed. (Ignore the code from the previous part.)

```
rose.add_visit("lucky bee")
daisy.add_visit("small bee")
rose.add_visit("cute bee")
daisy.add_visit("happy bee")
rose.add_visit("lucky bee")
```

- ☐ `daisy.bee_visits == rose.bee_visits`
- ■ `len(daisy.bee_visits) == 2`
- ☐ `len(Flower.bee_visits) == 4`
- ■ `len(daisy.bee_visits) == len(rose.bee_visits)`
- ■ `rose.bee_visits == ["lucky bee", "cute bee"]`
- ☐ `rose.bee_visits == ["cute bee", "lucky bee"]`

(d) **(1.0 pt)** What is the intended purpose of the `add_visit` method in the Flower class? Use no more than 2 sentences. (Consider both lines of `add_visit`.)

For full points, students must mention both:

- Tracks which bees have visted each flower
- Does not track duplicates

(e) **(1.0 pt)** Select the output displayed by the interactive Python interpreter when the code below is evaluated.

```
>>> x = 5 // 10 or 100 % 2 and -45
>>> y = [1, 2, 3].append(4) or [1, 2, 3].pop()
>>> print(x, y)
```

- ● 0 3
- ○ -45 3
- ○ -45 0
- ○ 0 0

(f) **(1.0 pt)** Select the output displayed by the interactive Python Interpreter when the code below is evaluated. The output may have multiple lines.

```
>>> def hola(ciao, hello):
...     print(ciao[0:2])
...     ciao = ciao[0][0]
...     def bonjour(bye):
...         print('holaaa')
...         return hello
...     return bonjour("yum")
>>> hi = hola([[1], 2, 3], "delish")
```

- ○ Line 1: `Error` Line 2: `yum`
- ● Line 1: `[[1], 2]` Line 2: `holaaa`
- ○ Line 1: `[[1], 2, 3]` Line 2: `delish`
- ○ Line 1: `[3, 2, [1]]`

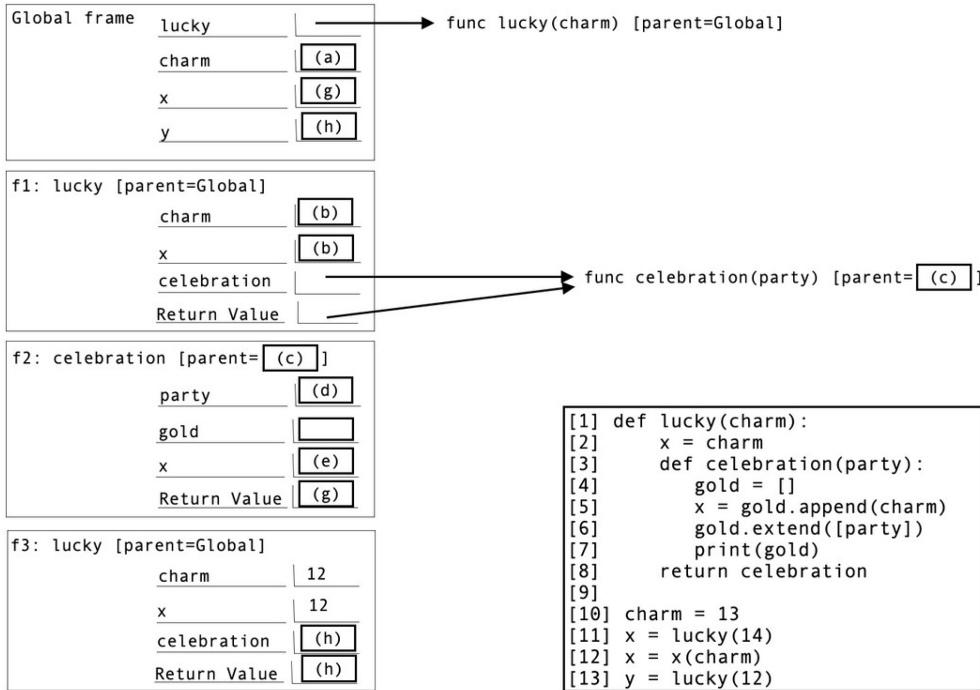
(g) **(1.0 pt)** Select the output displayed by the interactive Python Interpreter when the code below is evaluated. **Note that the `hola` function definition and `hi = ...` lines are identical to the previous subpart.**

```
>>> def hola(ciao, hello):
...     print(ciao[0:2])
...     ciao = ciao[0][0]
...     def bonjour(bye):
...         print('holaaa')
...         return hello
...     return bonjour("yum")
>>> hi = hola([[1], 2, 3], "delish")
>>> hi
```

- ○ `None`
- ● `'delish'`
- ○ `delish`
- ○ `holaaa`

2. (8.0 points) Lucky Charm

Complete the environment diagram below, then answer the corresponding questions. Each labeled blank (e.g., "(a)") has a related question. If two blanks share the same label, they have the same answer. Blanks without labels do not have associated questions and are not scored. If a blank contains an arrow pointing to a function, write the function exactly as it appears in the diagram.



(a) (1.0 pt) Fill in blank (a)

- ○ `None`
- ● `13`
- ○ `12`
- ○ `func lucky(charm) [parent=Global]`

(b) (1.0 pt) Fill in blank (b)

- ○ `func celebration(party) [parent=Global]`
- ○ `func lucky(charm) [parent=Global]`
- ○ `13`
- ○ `12`
- ● `14`

(c) (1.0 pt) Fill in blank (c)

- ○ `Global`
- ● `f1`
- ○ `f3`
- ○ `f4`

(d) (1.0 pt) Fill in blank (d)

- ○ `func celebration(party) [parent=Global]`
- ○ `func lucky(charm) [parent=Global]`
- ● `13`
- ○ `12`
- ○ `14`
- ○ `None`

(e) (1.0 pt) Fill in blank (e)

- ○ `func celebration(party) [parent=Global]`
- ○ `func lucky(charm) [parent=Global]`
- ○ `13`
- ○ `12`
- ○ `14`
- ● `None`

(f) (1.0 pt) There is no blank (f), please disregard this question (everyone received credit for this question).

- ● `func celebration(party) [parent=Global]`
- ● `func lucky(charm) [parent=Global]`
- ● `13`
- ● `12`
- ● `14`
- ● `None`

(g) (1.0 pt) Fill in blank (g)

- ○ `func celebration(party) [parent=Global]`
- ○ `func lucky(charm) [parent=Global]`
- ○ `13`
- ○ `12`
- ○ `14`
- ● `None`

(h) (1.0 pt) Fill in blank (h)

- ○ `func celebration(party) [parent=Global]`
- ○ `func lucky(charm) [parent=Global]`
- ○ `func celebration(party) [parent=f1]`
- ○ `func lucky(charm) [parent=f1]`
- ○ `func celebration(party) [parent=f2]`
- ○ `func lucky(charm) [parent=f2]`
- ● `func celebration(party) [parent=f3]`
- ○ `func lucky(charm) [parent=f3]`

3. (5.0 points) Blind Box Competition

In a **Blind Box Competition**, players open a series of boxes containing an item with varying rarity, which is either `"rare"` or `"common"`.

- A competition consists of multiple rounds. Each round has a round index `r` starting at 0.
- Each round has a limit `max_boxes` on the number of boxes that can be opened. Each box opened in a given round has a box index `b` starting at 0. The box index resets at the start of each round.
- A round ends once the first `"rare"` box is opened, or the `max_boxes` limit is reached (whichever comes first).
- A round is considered a win if at least one `"rare"` box is opened.

Implement the function `lucky_win`, which takes in three arguments:

- `rounds` (`int`): The number of rounds in the competition.
- `max_boxes` (`int`): The maximum number of boxes that can be opened **per round**.
- `open_box` (`function`): A function that takes in the current round index `r` and the current box index `b`, and returns a string representing the box's rarity (either `"rare"` or `"common"`).

The function should return a list `[rounds_won, total_boxes_opened, longest_streak]` where:

- `rounds_won` is the number of rounds won.
- `total_boxes` is the total number of boxes opened across **all rounds**.
- `longest_streak` is the length of the longest sequence of consecutive winning rounds.

```
def lucky_win(rounds, max_boxes, open_box):
    """
    >>> lucky_win(3, 1, lambda r, b: "rare")
    [3, 3, 3]
    >>> lucky_win(3, 1, lambda r, b: "common")
    [0, 3, 0]
    >>> lucky_win(3, 3, lambda r, b: "rare" if b == 1 else "common")
    [3, 6, 3]
    >>> lucky_win(3, 3, lambda r, b: "rare" if r == 2 and b == 2 else "common")
    [1, 9, 1]
    >>> lucky_win(5, 1, lambda r, b: "rare" if r in [0, 1, 2, 4] else "common")
    [4, 5, 3]
    """
    rounds_won, total_boxes, longest_streak, current_streak = 0, 0, 0, 0
    for ___(a)___:
        boxes_opened = 0
        found_rare = False

        for ___(b)___:
            box = open_box(r, b)
            boxes_opened += 1
            if box == "rare":
                found_rare = True
                break

        if ___(c)___:
            rounds_won += 1
            current_streak += 1
        else:
            current_streak = 0

        longest_streak = ___(d)___
        total_boxes += ___(e)___
    return [rounds_won, total_boxes, longest_streak]
```

(a) (1.0 pt) Fill in blank (a).

```
r in range(rounds)
```

(b) (1.0 pt) Fill in blank (b).

```
b in range(max_boxes)
```

(c) (1.0 pt) Fill in blank (c).

```
found_rare
```

(d) (1.0 pt) Fill in blank (d).

Hint: What built-in Python function should you call here?

```
max(longest_streak, current_streak)
```

(e) (1.0 pt) Fill in blank (e).

```
boxes_opened
```

4. (8.0 points) Attention Trader Joe's Shoppers

You're shopping at Trader Joe's and notice that there are some items on sale! Implement the function `apply_discount` which takes in `discount` (a float from 0.0 to 1.0) and `discounted_items` (a list of item names that are on sale). For example, if `discount` is 0.4 that means that all the items in the list are 40% off.

`apply_discount` should return a function `update_prices`, which takes in a shopper's `cart` (represented as a dictionary mapping item names to their regular prices) and applies the `discount` to each of the applicable items by **modifying the cart**. Additionally, `update_prices` returns the number of items that the discount was applied to.

```
def apply_discount(discount, discounted_items):
    """
    >>> cart1 = {
    ...     'everything bagel seasoning': 3.0,
    ...     'mini ice cream cones': 5.0,
    ...     'chocolate peanut butter cups': 4.0
    ... }
    >>> cart2 = {'everything bagel seasoning': 3.0}
    >>> on_sale = ['mini ice cream cones', 'chocolate peanut butter cups']
    >>> discount_func = apply_discount(0.2, on_sale)  # all of the sale items in a cart are 20% off
    >>> discount_func(cart1)
    2
    >>> cart1
    {'everything bagel seasoning': 3.0, 'mini ice cream cones': 4.0, 'chocolate peanut butter cups': 3.2}
    >>> discount_func(cart2)
    0
    >>> cart2
    {'everything bagel seasoning': 3.0}
    """
    def update_prices(cart):
        num_changed = 0
        for item in cart:
            if item in ___(a)___:
                ___(b)___
                ___(c)___
        return ___(d)___
    return ___(e)___
```

(a) (1.0 pt) Fill in blank (a).

```
discounted_items
```

(b) (1.0 pt) Fill in blank (b).

```
cart[item] *= 1 - discount
```

(c) (1.0 pt) Fill in blank (c).

```
num_changed += 1
```

(d) (1.0 pt) Fill in blank (d).

```
num_changed
```

(e) **(1.0 pt)** Fill in blank (e).

update_prices

(f) **(1.0 pt)** Suppose we move `num_changed = 0` outside of `update_prices` to the `apply_discount` function but keep the rest of the code the same. How would the program change, if at all? The updated code is restated below (but the correct implementation is hidden).

```
def apply_discount(discount, discounted_items):
    num_changed = 0  # new location
    def update_prices(cart):
        for item in cart:
            ...
    return ...
```

- ● The program would throw an `UnboundLocalError` since `num_changed` is defined outside our current frame.
- ○ The program would run the same way with no errors.
- ○ There would be no errors, but `num_changed` may have a different value at the end of the program.

(g) **(2.0 pt)** Select all of the following calls to `apply_discount` that would NOT result in an error after running the following code. Assume that `apply_discount` is implemented correctly.

```
on_sale = ['mini ice cream cones', 'chocolate peanut butter cups']
cart1 = {
    'everything bagel seasoning': 3.0,
    'mini ice cream cones': 5.0,
    'chocolate peanut butter cups': 4.0
}
cart2 = {'everything bagel seasoning': 3.0}
```

- ■ `apply_discount(0.2, on_sale)(cart1)`
- ■ `apply_discount(0.2, on_sale)({})`
- □ `apply_discount(0.2, on_sale)(cart1)(cart2)`
- □ `apply_discount(0.2, on_sale)()`
- □ `apply_discount()(cart1)`
- ■ `apply_discount(0.2, on_sale)`

5. (6.0 points) Unzip and Apply

(a) (3.0 pt) Implement the function `unzip`, which takes in a list of tuples `zipped_lst` and integer `group_index`, and retrieves the "unzipped" list with the given `group_index`. That is, we want to perform the inverse of the built-in `zip` function Python. See the doctests to better understand what is happening. You may assume each tuple in `zipped_lst` has the same length.

```
def unzip(zipped_lst, group_index):
    return [
        zipped_lst[tup_index][group_index]
        for tup_index in range(len(zipped_lst))
    ]
```

(b) Assume that you have a correct implementation of `unzip`. Now implement the `unzip_and_apply` function, which takes in a list of tuples `zipped_lst` and a function `func`, and returns a list `result`, which applies the function `func` to each unzipped list.

That is each item of `result` is the result of applying `func` to the corresponding unzipped list.

See the doctests to better understand what is happening. You may assume each tuple in `zipped_lst` has the same length.

```
def unzip_and_apply(zipped_lst, func):
    """
    >>> zips = [(1, 2), (3, 4), (5, 6)]
    >>> unzip(zips, 0) == [1, 3, 5] # sum([1, 3, 5]) == 9
    True
    >>> unzip(zips, 1) == [2, 4, 6] # sum([2, 4, 6]) == 12
    True
    >>> unzip_and_apply(zips, sum)
    [9, 12]
    >>> zips = [(1, 2), (3, 4), (5, 6)]
    >>> unzip_and_apply(zips, min)
    [1, 2]
    >>> zips = [('a', 'b', 'c'), ('d', 'e', 'f')]
    >>> unzip(zips, 0) == ['a', 'd'] # min(['a', 'd']) == 'a'
    True
    >>> unzip_and_apply(zips, min)
    ['a', 'b', 'c']
    """
    result = []
    num_groups = ____(a)____
    for group_index in range(num_groups):
        group_values = ____(b)____
        result.append(____(c)____)
    return result
```

i. (1.0 pt) Fill in blank (a).

```
len(zipped_lst[0])
```

ii. (1.0 pt) Fill in blank (b).

```
unzip(zipped_lst, group_index)
```

iii. (1.0 pt) Fill in blank (c).

```
func(group_values)
```

6. (8.0 points) Recursive Palindrome

Edwin and Reema were particing their recursion skills and wanted to come up with some lists which look Recursive. They then thought about palindromes. A palindrome is a string or number that reads the same forwards and backwards. For example, `"racecar"` and `88` are palindromes, while `"hello"` and `21` are not.

However, what if we had a `list` as a palindrome? We are going to write a function `is_symmetrical` that determines whether a given list is a palindrome using recursion.

```
def is_symmetric(lst):
    """
    >>> is_symmetric([])
    True
    >>> is_symmetric(["+", "+"])
    True
    >>> is_symmetric([1, 2, 3, 2, 1])
    True
    >>> is_symmetric(["hello", "hello"])
    True
    >>> is_symmetric([1, 2, 3, 4, 5])
    False
    """
    def check_symmetry(start, end):
        if start > end:
            return True
        if _____(a)_____:
            return True
        if _____(b)_____:
            return False
        return ____(c)____
    return check_symmetry(0, len(lst) - 1)
```

(a) (1.0 pt) Fill in blank (a).

```
start == end
```

(b) (1.0 pt) Fill in blank (b).

```
lst[start] != lst[end]
```

(c) (3.0 pt) Fill in blank (c).

```
check_symmetry(start + 1, end - 1)
```

(d) (1.0 pt) The `is_symmetric` function works pretty well, but it's not perfect. Some inputs will cause the function to throw an error. Provide an input that will cause the function to throw an error. You may consider all possible inputs, including unexpected types of values.

```
There are many possible solutions, one example is: is_symmetric(5).
See explanation below.
```

(e) **(2.0 pt)** Based on the input you provided, describe the type of error that would occur. Then describe one possible way to fix the error. Use no more than 3 sentences.

If you happen to remember the exact name of Python's error message, you can include it in your answer, but it is not required.

There are many valid solutions:

- Any number, boolean, dict is invalid because you cannot index into it. The fix is to return early or coerce the value to a list (or string).
- Note that a string is actually a valid input (because you can index into strings) and thus does not cause an error.

7. (8.0 points) Escape Lumon

Helly is a new employee at Lumon Industries. She is trying to find her way out of her office building, but she's locked in! To get out, she will need to collect at least 3 security badges.

Luckily, her coworker Mark has given her a grid map of the building's hallways, implemented as a Python nested list (list of lists). Each cell in the grid has a 0 or 1, with a 1 representing a location where she can get a security badge.

Implement the function `count_paths`, which takes in a `grid` and returns the number of paths Helly can take from the top left corner of the grid to the bottom right corner of the grid, only using movements to the right or down. Remember that a valid path must have Helly collect at least 3 security badges before reaching the end.

```
def count_paths(grid):
    """
    Counts the number of distinct paths from the top left to bottom right of a grid
    while collecting at least 3 security badges.

    >>> simple_grid = [
    ...     [1, 0, 0],
    ...     [1, 0, 0],
    ...     [1, 0, 0],
    ... ]
    >>> count_paths(simple_grid)
    1

    >>> # R = right, D = down
    >>> # Path 1: RRDD, Path 2: RDRD, Path 3: RDDR, Path 4: DRRD, Path 5: DRDR
    >>> complex_grid = [
    ...     [1, 0, 1],
    ...     [0, 1, 0],
    ...     [0, 0, 1],
    ... ]
    >>> count_paths(complex_grid)
    5

    >>> zeros_grid = [
    ...     [0, 0, 0],
    ...     [0, 0, 0],
    ...     [0, 0, 0],
    ... ]
    >>> count_paths(zeros_grid)
    0
    """
    def helper(row, col, badges):
        if row >= len(grid) or col >= len(grid[0]):
            return ___(a)___
        badges += ___(b)___
        if ___(c)___:
            if badges >= 3:
                return ___(d)___
            else:
                return ___(e)___
        return ___(f)___
    return ___(g)___
```

(a) **(1.0 pt)** Fill in blank (a).

- ○ -1
- ● 0
- ○ 1
- ○ 3

(b) **(1.0 pt)** Fill in blank (b).

```
grid[row][col]
```

(c) **(1.0 pt)** Fill in blank (c).

```
row == len(grid) - 1 and col == len(grid[0]) - 1
```

(d) **(1.0 pt)** Fill in blank (d).

- ○ -1
- ○ 0
- ● 1
- ○ 3

(e) **(1.0 pt)** Fill in blank (e).

- ○ -1
- ● 0
- ○ 1
- ○ 3

(f) **(2.0 pt)** Fill in blank (f).

```
helper(row, col + 1, badges) + helper(row + 1, col, badges)
```

(g) **(1.0 pt)** Fill in blank (g) with the correct code and reasoning.

- ● `helper(0, 0, 0)` because Helly starts at the top left and with 0 security badges
- ○ `helper(0, 0, 3)` because Helly starts at the top left and needs 3 security badges
- ○ `helper(len(grid) - 1, len(grid[0]) - 1, 0)` because Helly's destination is the bottom right and she starts with 0 security badges
- ○ `helper(len(grid) - 1, len(grid[0]) - 1, 3)` because Helly's destination is the bottom right and she needs 3 security badges

8. (0.0 points) Just for fun...

Draw something fun, or write a message for the staff!

(a)