DS C88C Fall 2023 Computational Structures in Data Science

MIDTERM

INSTRUCTIONS

- **Do NOT open the exam until you are instructed to do so!**
- You **must not** collaborate with anyone inside or outside of C88C.
- You **must not** use any internet resources to answer the questions.
- If you are taking an online exam, at this point you should have started your Zoom / screen recording. If something happens during the exam, focus on the exam! Do not spend more than a few minutes dealing with proctoring.
- When a question specifies that you must rewrite the completed function, you should **not** recopy the doctests.
- The exam is closed book, closed computer, closed calculator, except your hand-written 8.5" x 11" cheat sheets of your own creation and the official C88C Reference Sheet

Full Name	
Student ID Number	
Official Berkeley Email (@berkeley.edu)	
What room are you in?	
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.
- For fill-in-the-blank coding problems, we will only grade work written in the provided blanks.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.
  - **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) One Small List

Consider the list `small_list`. Complete the following questions related to this list. You may not modify this list in any way unless explicitly stated in the question. Each part in this section is **independent**, meaning that any modifications you may make in particular part **will not be saved** between parts.

```
>>> small_list = [
    [41, 70, 69],
    [
        [92, 17],
        [
            [35, 85, 67],
            [
                [24, 3],
                [46, 74],
            ],
            [19, 2],
        ],
        [15, 11, 8],
    ],
    [
        [
            [59, 47],
            [68, 0],
        ],
        [80, 3, 1],
    ],
    [97, 2]
]
```

(a) (1.0 pt) Fill in the blanks such that this expression evaluates to 0. You may not use any list functions (len, min, max, sum) in your answer.

```
>>> small_list_____________________________
```

(b) (2.0 pt) What is the result of the following expression?

```
[ x[0] for x in small_list[2] ]
```

(c) (2.0 pt) Add a **one element list**, [5], to the sublist [15, 11, 8]. The resulting sublist should look like the following:

```
[15, 11, 8, [5]]
```

(d) (1.0 pt) Which of the following indexing operations will return the number 3? Select all that apply.

- ◯ `small_list[2][0][1][1]`
- ◯ `small_list[2][1][1][1]`
- ◯ `small_list[2][1][1]`
- ◯ `small_list[1][1][1][0][1]`
- ◯ `small_list[1][1][1][0]`

(e) (1.0 pt) What is a result of the following expression?

```
small_list[1][3]
```

◯ `[80, 3, 1]`

◯ `Error`

◯ `None`

◯ `[15, 11, 8]`

◯ `0`

2. (7.0 points) What Would Python Do (WWPD)

For each expression 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" (if any lines are displayed before the error, include those in your output). If a function is returned, write "Function". If the value "None" is returned, write "None".

NOTE: Assume each part is executed *in order*. Previous lines DO impact the current expression. (i.e., part B assumes part A was executed, as so on.)

```
r = 4
z = [7]
g = lambda x: x*3
dct = {'evens': [lambda x,y: x+y], 'odds': [lambda: print('hi')] }

def huh(dct, f):
    even_len = len(dct['evens'])
    odd_len = len(dct['odds'])
    curr_len = even_len + odd_len
    if curr_len % 2 == 0:
        dct['evens'].append(f)
    else:
        dct['odds'].append(f)
    return dct['evens'] + dct['odds']
```

(a) (2.0 pt)

```
>>> a = z.extend([r])
>>> a, z
```

- ◯ `None, [7, 4]`
- ◯ `None, None`
- ◯ `[7, [4]], [7, [4]]`
- ◯ `[7, [4]], None`
- ◯ `None, [7, [4]]`
- ◯ `[7, 4], [7, 4]`
- ◯ `Error`
- ◯ `[7, 4], None`

(b) (1.0 pt)

```
>>> a = lambda x: x*x
>>> a
```

(c) (1.0 pt)

```
>>> a(z[0])
```

(d) (1.0 pt)

```
>>> dct['evens'][0](r, g(r))
```

(e) (1.0 pt)

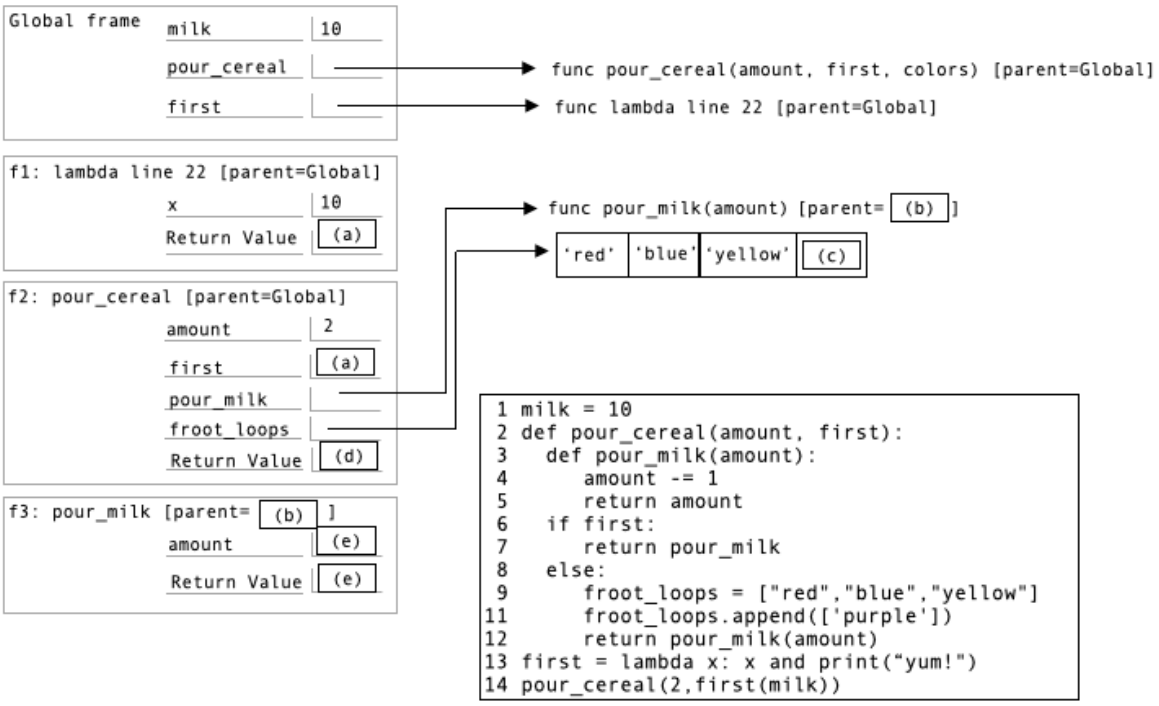
```
>>> q = huh(dct, a)
>>> q[1](3)
```

(f) (1.0 pt)

```
>>> w = huh(dct, g)
>>> w[2](3)
```

3. (5.0 points) Milk Or Cereal First?

Fill in the blanks to complete the environment diagram. All the code used is in the box to the right, and the code runs to completion.



(a) (1.0 pt) What is the return value of the lambda function in frame `f1`? (box a)

- ◯ "Yum!"
- ◯ False
- ◯ True
- ◯ None

(b) (1.0 pt) What is the parent frame of the function `pour_milk` in f3? (box b)

(c) (1.0 pt) What is the 4th elemnt of `froot_loops`? (box c)

(d) (1.0 pt) What is the return value of `pour_cereal` ? (box d)

(e) (1.0 pt) What is the value of `amount` ? (box e)

4. (8.0 points) A Product of Your Imagination

George would like to take the dot product of two Python lists. The dot product is defined as the sum of the elementwise products of two lists. In other words, we pair up either list element-by-element, in order, and multiply each element pair. We then sum each element pair product and return the sum. Both lists are assumed to be of the same length.

The procedure is as follows:

Take the elementwise product of each of the entries between both lists Sum each of the products found in step 1 Return the sum

```
def dot_prod(lst1, lst2):
    ...
    return ...

>>> dot_prod([1, 2, 3, 4], [10, 20, 30, 40])
300

# (1 * 10) + (2 * 20) + (3 * 30) + (4 * 40) =
# 10 + 40 + 90 + 160 =
# 300
```

For each of the code blocks below, select the option that best fits what would occur if the code were executed by Python. Each code block either runs without error and as intended, runs without error, but not as intended, or errors and does not run at all. If the code runs without error, but not as intended above, or if the code errors and does not run at all, then explicitly state what line the error is at and describe what went wrong. If the code runs without error and as intended, leave the space blank.

Note: In each of the incorrect code blocks below there is one, and only one line that is erroneous and if this line were to be corrected, the function would run error-free and as intended above. **Note:** The call to the `dot_prod` function can be erroneous.

(a) (2.0 pt) Select the option which describes the result of this code, then in the next part describe your response if necessary.

```
(1) def dot_prod(lst1, lst2):
(2)     return sum([lst1[i] * lst2[i] for i in range(4)])
(3)
(4) dot_prod([2, 3, 2, 1, 4], [10, 20, 30, 40, 50])
```

◯ The code errors and does not run.

◯ The code runs without error, but not as intended above.

◯ The code runs without error and as intended above.

(b) If code the runs without error, but not as intended above, or if the code errors and does not run at all, then explicitly state what line the error is at and describe what went wrong. If the code runs without error and as intended above, leave the space blank.

(c) (2.0 pt) Select the option which describes the result of this code, then in the next part describe your response if necessary.

```
(1) def dot_prod(lst1):
(2)     def fn(lst2):
(3)         return sum([lst1[i] * lst2[i] for i in range(len(lst1))])
(4)     return fn
(5)
(6) dot_prod([2, 3, 2, 1, 4], [10, 20, 30, 40, 50])
```

◯ The code runs without error, but not as intended above.

◯ The code runs without error and as intended above.

◯ The code errors and does not run.

(d) If code the runs without error, but not as intended above, or if the code errors and does not run at all, then explicitly state what line the error is at and describe what went wrong. If the code runs without error and as intended above, leave the space blank.

(e) (2.0 pt) Select the option which describes the result of this code, then in the next part describe your response if necessary.

```
(1) def dot_prod(lst1, lst2):
(2)     s = 0
(3)     for i in lst1:
(4)         s += lst1[i] * lst2[i]
(5)     return s
(6)
(7) dot_prod([2, 3, 2, 1, 4], [10, 20, 30, 40, 50])
```

◯ The code block runs without error, but not as intended above.

◯ The code block runs without error and as intended above.

◯ The code block errors and does not run.

(f) If code the runs without error, but not as intended above, or if the code errors and does not run at all, then explicitly state what line the error is at and describe what went wrong. If the code runs without error and as intended above, leave the space blank.

(g) (2.0 pt) Select the option which describes the result of this code, then in the next part describe your response if necessary.

```
(1) def dot_prod(lst1, lst2):
(2)     lst = list(zip(lst1, lst2))
(3)     fn = lambda tup: tup[-1] * tup[-2]
(4)     return sum(list(map(fn, lst)))
(5)
(6) dot_prod([2, 3, 2, 1, 4], [10, 20, 30, 40, 50])
```

Just as a reminder, this is what the zip function does:

```
>>> list(zip([0, 1, 2, 3, 4], [10, 20, 30, 40, 50]))
[(0, 10), (1, 20), (2, 30), (3, 40), (4, 50)]
```

◯ The code block runs without error and as intended above.

◯ The code block runs without error, but not as intended above.

◯ The code block errors and does not run.

(h) If code the runs without error, but not as intended above, or if the code errors and does not run at all, then explicitly state what line the error is at and describe what went wrong. If the code runs without error and as intended above, leave the space blank.

5. (8.0 points) No More Couting Dollars, We'll Be Counting Chars

Given a string `word`, count the number of times each character is used within `word` in the form of a dictionary with its key-value pair being `character: number of occurances`. Each count will start with an empty dictionary. Assume all `word`s will be lowercase and only use the alphabet. Look at the doctests for help!

(a) (4.0 pt) First, we need to create a function that returns a dictionary of the counts of each mention of a character in the word.

```
def counting_chars(word):
    """
    >>> word1 = 'hi'
    >>> word2 = 'hello'
    >>> chars1 = counting_chars(word1)
    >>> chars2 = counting_chars(word2)
    >>> chars1
    {'h': 1, 'i': 1}
    >>> chars2
    {'h': 1, 'e': 1, 'l': 2, 'o': 1}
    """
    chars = {}
    for character in word:
        if character _____________________:
            ______________________________
        else:
            ______________________________
    return _______________________________
```

```
'''
def counting_chars(word):
    counts =
    for character in word:

        if character ________________________________________________:

            ____________________________________________________________
        else:

            ____________________________________________________________

    return ____________________________________________________________
'''
```

(b) (4.0 pt) Now, we would like to remove all mentions of any vowels within the dictionaries and return a dictionary that only has counts and mentions of consonants. (Hint: dictionaries can be written in the format of list comprehensions using key:value as its output)

```
def remove_vowels(chars):
    """
    >>> word1 = 'hi'
    >>> word2 = 'hello'
    >>> chars1 = counting_chars(word1)
    >>> chars2 = counting_chars(word2)
    >>> chars1
    {'h': 1, 'i': 1}
    >>> chars2
    {'h': 1, 'e': 1, 'l': 2, 'o': 1}
    >>> consonants1 = remove_vowels(chars1)
    >>> consonants2 = remove_vowels(chars2)
    >>> consonants1
    {'h': 1}
    >>> consonants2
    {'h': 1, 'l': 2}
    """
    vowels = 'aeiou'
    return { ________:________ for key, value in ________________ if __________ not in vowels }
```

```
‘‘‘
def remove_vowels(chars):
    vowels = ’aeiou’

    return { ______________________ :  ______________________

        for key, value in _________________________

        if ______________________ not in vowels }
‘‘‘
```

6. (8.0 points) List Finder

At PiazzaPizza, they have a special discount that is applied when a customer orders **BOTH a drink AND food key** from their menu. Customers also can write their own request form which gets populated when they order an key that doesn't exist on the menu. Help them implement `customer`, a function that takes in a specified discount function `discount_func` and returns an `order` function that allows each customer to order a `food` and `drink` from a predetermined `FOOD_MENU` and `DRINKS_MENU`.

The function `order` should:

check if both the `food` and `drink` key that the customer ordered is in the menu. If **BOTH** the `food` and the `drink` keys exist, apply `discount_func` to the *sum* of keys' cost (stored in the 2 dictionaries: `FOOD_MENU` and `DRINKS_MENU`) and return the total cost of the order **after** the discount function is applied. If only the `food` **OR** `drink` key exists, return the cost of that food/drink. If **NEITHER** of `food` nor `drink` is in the restruant menu, add the keys to the customers' `request` list and return the `request` list.

You do not need to consider the case where one key is in the menu and the other is not.

```
FOOD_MENU = {"burger": 10, "pizza": 15}
DRINKS_MENU = {"lemonade": 5, "milkshake": 7}
def customer(discount_func):
    """
    >>> discount_function = lambda x, y: 0.5
    >>> c = customer(discount_function)
    >>> c("burger", "lemonade")
    7.5    # (10 + 15) * 0.5
    >>> c("burger", None)
    10    # 10
    >>> c(None, "lemonade")
    5     # 5
    >>> c("fries", "coke")
    ["fries", "coke"]     # not in the menus
    >>> c("tater tots", "water")
    ["fries", "coke", "tater tots", "water"]     # not in the menus
    """
    def customer(discount_func):
        requests = []
        def order(food, drink):
            if ____(a)____:
                food_cost = FOOD_MENU[food]
                drink_cost = DRINKS_MENU[drink]
                discount = ____(b)____
                return ___(c)___
            elif food in FOOD_MENU:
                return FOOD_MENU[food]
            elif drink in DRINKS_MENU:
                return DRINKS_MENU[drink]
            else:
                ___(d)___
                ___(e)___

        return ___(f)___
```

(a) (2.0 pt) Write an expression that completes blank (a)

(b) (2.0 pt) Write an expression that completes blank (b)

(c) (1.0 pt) Write an expression that completes blank (c)

(d) (1.0 pt) Select the expression that fills in blank (d)

- ◯ `requests+=1`
- ◯ `requests.append((food, drink))`
- ◯ `requests.extend([food, drink])`
- ◯ `requests[food]`
- ◯ `requests.append([food, drink])`

(e) (1.0 pt) Select the expression that fills in blank (e)

- ◯ `requests`
- ◯ `discount_func`
- ◯ `discount`
- ◯ `customer`
- ◯ `order`

(f) (1.0 pt) Select the expression that fills in blank (f)

- ◯ `discount`
- ◯ `discount_func`
- ◯ `customer`
- ◯ `order`
- ◯ `requests`

7. (5.0 points) Shopping List

You are going to buy groceries at the store. You have written a shopping list that is a `list` of `dictionaries`, where each dictionary is of the following format:

`{"key": "keyname", "amount": amount, "price": price}`

`"key"`, `"amount"`, and `"price"` are strings, and they refer to the specific key being bought (string), the amount of each key (int), and the price per key (int), respectively. You want to aggregate the total price for each key being bought. Define a function that takes in a list and returns a dictionary of the following format:

`{"keyname": totalprice}`

where `"keyname"` refers to the name of the key and `totalprice` refers to the total price of the keys. You may assume that the list passed in will be of the specified format.

```
def aggregate_price(lst):
    """
    >>> shopping_list = [
                          {"key": "apple", "amount": 3, "price": 5},
                          {"key": "banana", "amount": 6, "price": 2},
                          {"key": "milk", "amount": 1, "price": 3},
                          {"key": "carrots", "amount": 3, "price": 5},
                          {"key": "apple", "amount": 2, "price": 5},
                          {"key": "banana", "amount": 8, "price": 2},
                          {"key": "apple", "amount": 4, "price": 5}
                        ]
    >>> res = aggregate_price(shopping_list)
    >>> res
    {"apple": 45, "banana": 28, "milk": 3, "carrots": 15}
    """
    agg_dict = {}
    for d in lst:
      if __________(a)__________:
        __________(b)__________
      else:
        __________(c)__________
    return agg_dict
```

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

(b) (2.0 pt) Fill in blank (b).

(c) (2.0 pt) Fill in blank (c).

8. (12.0 points) What are you doing, Mr. Robot?

Elliot is a hacker who is trying to corrupt the files stored on Evil Corp's servers. All files on the server belong to one directory, called the *root directory*. The root directory is represented as a dictionary, where each key is a file name **or** directory name.
If the dictionary entry represents a file, its value will be a string representing the contents of the file. If the dictionary entry represents a directory, its value will be another dictionary with the same format as the root directory dictionary.

Elliot needs your help to write a Python function `corrupt_files`, which takes in a dictionary, `root` (the root directory), and integer `depth`. The function should **replace** the contents of all files with the string `"mrrobot"` and return the total number of files that were num$_c$*hanged*.

```
>>> data = {'hi': 5}
>>> type(data) == dict
True
>>> type('evil.txt') == str
True
```

```
def corrupt_files(root, depth):
    """
    >>> root0 = {}
    >>> root1 = {
    ...     "evil.txt": "a",
    ...     "corp.txt": "b",
    ...     "allsafe.txt": "c"
    ... }
    >>> root2 = {
    ...     "src": {
    ...         "hi.pdf": "hello, Giddeon"
    ...     },
    ...     "empty_dir": {}
    ... }
    >>> root3 = {
    ...     "hello.txt": "hello world",
    ...     "c88c": {
    ...         "README.md": "text",
    ...         "labs": {
    ...             "lab00.py": "code",
    ...             "lab01.py": "more code"
    ...         },
    ...         "hw": {
    ...             "hw00.py": "even more code",
    ...             "notes": {
    ...                 "lec00.txt": "lecture notes"
    ...             }
    ...         }
    ...     },
    ...     "bye.pdf": "farewell"
    ... }
    >>> corrupt_files(root0, 0)
    0
    >>> root0
    {}
    >>> corrupt_files(root1, 1)
    3
    >>> root1
    {'evil.txt': 'mrrobot', 'corp.txt': 'mrrobot', 'allsafe.txt': 'mrrobot'}
    >>> corrupt_files(root2, 2)
    1
    >>> root2
    {'src': {'hi.pdf': 'mrrobot'}, 'empty_dir': {}}
```

```
>>> corrupt_files(root3, 1)
2
>>> root3
{
    "hello.txt": "mrrobot",
    "c88c": {
        "README.md": "text",
        "labs": {
            "lab00.py": "code",
            "lab01.py": "more code"
        },
        "hw": {
            "hw00.py": "even more code",
            "notes": {
                "lec00.txt": "lecture notes"
            }
        }
    },
    "bye.pdf": "mrrobot"
}
>>> corrupt_files(root3, 3)
4
>>> root3
{
    "hello.txt": "mrrobot",
    "c88c": {
        "README.md": "mrrobot",
        "labs": {
            "lab00.py": "mrrobot",
            "lab01.py": "mrrobot"
        },
        "hw": {
            "hw00.py": "mrrobot",
            "notes": {
                "lec00.txt": "lecture notes"
            }
        }
    },
    "bye.pdf": "mrrobot"
}
"""
if _________________________ (a) ___________________________________:
    return 0
num_changed = ____________________ (b) _________________________________
for key in root:
    if _______________ (c) _______ and __________ (d) __________:
        root[key] = ___________________ (e) ______________________
        num_changed += _______________ (f) ________________________
    elif ____________________ (g) _____________________________:
        num_changed += _________________ (h) ________________________
return num_changed
```

1. **(1.0 pt)** Which of the following could fill in blank (a)?

   ◯ `depth == 0`

   ◯ `"mrrobot" in root`

   ◯ `not root`

   ◯ `depth != len(root)`

2. (1.0 pt) Fill in blank (b).

3. (1.0 pt) Which of the following could fill blank (c)?

- ◯ `type(root[key]) == dict`
- ◯ `type(key) == str`
- ◯ `type(root[key]) == str`
- ◯ `"mrrobot" not in root`

4. (1.0 pt) Which of the following could fill blank (d)?

- ◯ `"mrrobot" not in root`
- ◯ `key != "mrrobot"`
- ◯ `root[key] != "mrrobot"`
- ◯ `root[key] == "mrrobot"`

5. (2.0 pt) Fill in blank (e).

6. (2.0 pt) Fill in blank (f).

7. (2.0 pt) Fill in blank (g).

8. (2.0 pt) Fill in blank (h).

No more questions.