# DS C88C Spring 2024 Computational Structures in Data Science

MIDTERM <NAME>

**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. (4.0 points) Python Potpourri**

**(a) (1.0 pt)** Suppose we have run the following code:

```
lst = [1, 2, 3]
```

Which of the following are equivalent to `lst.append(4)`? Select all that apply.

☐ `lst += [4]`

☐ `lst.extend([4])`

☐ `lst.append([4])`

☐ `lst += 4`

☐ `lst.extend(4)`

**(b) (1.0 pt)** Suppose we run the code below:

```
novels = {
    'Six of Crows': ['Kaz', 'Inej', 'Jesper'],
    'All My Rage': ['Sal', 'Noor'],
    'The Lightning Thief': ['Percy', 'Annabeth', 'Grover']
}

for title in novels:
    novels[title] = [ len(c) for c in novels[title] ]

mystery = [ novels[title][1] for title in novels ]
```

What is the value of `mystery`? If you think the code would cause an error, select "The code would error".

◯ `['Six of Crows', 'All My Rage', 'The Lightning Thief']`

◯ `[4, 4, 8]`

◯ `['i', 'l', 'h']`

◯ `[3, 3, 5]`

◯ `[['a', 'n', 'e'], ['a', 'o'], ['e', 'n', 'r']]`

◯ The code would error

**(c) (1.0 pt)** Suppose we run the following code:

```
lst = [87, 45, 98]
total = 0
```

**True or False**: The 2 blocks of code below are equivalent. (Assume they run independently of each other.)

```
# Block 1
i = 0
while i < len(lst):
    total += lst[i] + 87

# Block 2
for x in range(len(lst)):
    total = total + lst[x] + 87
```

◯ True

◯ False

**(d) (1.0 pt)** What is the value of `filtered` after the code below is run?

```
lst = ['Andor', 'Luthen', 'Mon', 'Brasso', 'Maarva', 'Bix', 'Dedra']
func = lambda x: len(x) % 2 == 0
filtered = list(filter(func, lst))
```

◯ `['Andor', 'Mon', 'Bix', 'Dedra']`

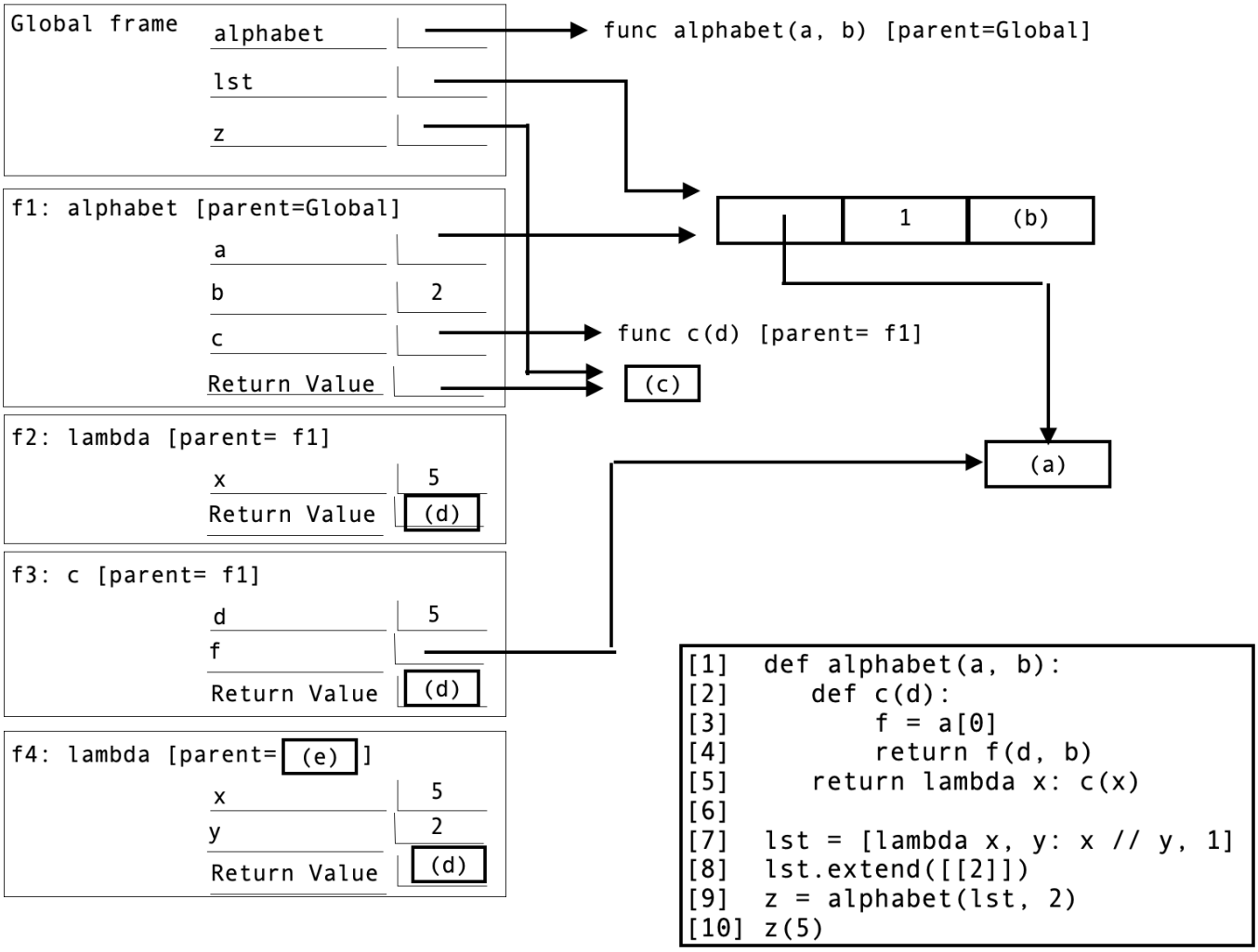
◯ `['Luthen', 'Brasso', 'Maarva']`

◯ `['Andor', 'Luthen', 'Mon', 'Brasso', 'Maarva', 'Bix', 'Dedra']`

◯ None of the above

**2. (5.0 points) Alphabet Galore!**

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 element at index 0 of the list `lst`? (box a)

- ◯ `func c(d)`
- ◯ `None`
- ◯ `2`
- ◯ `func (x, y) <line 7> [parent=Global]`

**(b) (1.0 pt)** What is the element at index 2 of the list `lst`? (box b)

**(c) (1.0 pt)** What is the return value of the function `alphabet`? (box c)

- ◯ `1`
- ◯ `f`
- ◯ `func (x, y) <line 7>`
- ◯ `func (x) <line 5> [parent=f1]`

**(d) (1.0 pt)** What is the return value of `lambda` in f2 and the return value of the function `c`? (box d)

**(e) (1.0 pt)** What is the parent function of the `f4` lamda function? (box e)

**3. (6.0 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.)

```
def numbers(one, two, three):
    for four, dog in one.items():
        two.append(dog(four))
        three[four] = len(one)
    return (two, three)

bell = {1: lambda x: x + 2, 2: lambda y: y - 3}
table = {'a': 1, 'b': 2}
lst = []

jazz = numbers(bell, lst, table)
```

**(a) (2.0 pt)**

```
>>> jazz[0]
```

**(b) (1.0 pt)**

```
>>> jazz[1]
```

**(c) (1.0 pt)**

```
>>> bell[1]()
```

**(d) (1.0 pt)**

```
>>> bell[1](table['a'])
```

**(e) (1.0 pt)**

```
>>> bird = [i ** 2 for i in table.values()][-1]
>>> bird
```

**4. (5.0 points) Daredevil**

Daredevil is a vigilante protecting the streets of New York City. The city is represented by a 2D list called `grid` with `n` rows and `n` columns, with each cell containing the number 0 (representing a house) or 1 (representing a skyscraper).

Each night, Daredevil starts at position `(0, 0)` (e.g. the top left corner) and ends at position `(n - 1, n - 1)` (e.g. the bottom right corner). **Note:** A skyscraper can be in any location in the city grid, including on the starting space and ending space.

Daredevil can only move right one cell or down one cell at a time. An optimal route includes climbing no more than 2 skyscrapers.

Rebecca has written a function (that may contain bugs) which returns the number of optimal routes Daredevil can take.

```
def num_optimal_routes(grid):
    """
    >>> grid0 = [
    ...     [0, 0],
    ...     [0, 0]
    ... ]
    >>> num_optimal_routes(grid0)
    2
    >>> grid1 = [
    ...     [0, 0],
    ...     [1, 0]
    ... ]
    >>> num_optimal_routes(grid1)
    2
    >>> grid2 = [
    ...     [1, 0],
    ...     [1, 1]
    ... ]
    >>> num_optimal_routes(grid2)
    1
    >>> grid3 = [
    ...     [1, 1],
    ...     [1, 1]
    ... ]
    >>> num_optimal_routes(grid3)
    0
    """
    def helper(row, col, num_skyscrapers):
        if num_skyscrapers > 2:
            return 1
        elif row == len(grid) - 1 and col == len(grid) - 1:
            if num_skyscrapers + grid[row][col] > 2:
                return 0
            return 1
        else:
            right = helper(row, col + 1, num_skyscrapers + grid[row][col])
            down = helper(row + 1, col, num_skyscrapers + grid[row][col])
            return right + down
    return helper(0, 0, 0)
```

**(a) (1.0 pt)** Consider this part of the function:

```
if num_skyscrapers > 2: # line 1
    return 1            # line 2
```

Select the action that would fix the bug in this part of the code, if a bug exists.

◯ Replace line 1 with `if num_skyscrapers >= 2`

◯ Replace line 2 with `return 0`

◯ Replace line 2 with `return 2`

◯ There are no bugs in this part of the code

**(b) (1.0 pt) True or False**: The code is missing a base case.

◯ True

◯ False

**(c) (2.0 pt)** If you selected True for the subpart above, explain briefly (1-2 sentences) what the missing base case is and what should be returned in this case. If you selected False, leave this part blank.

**(d) (1.0 pt)** Consider this part of the function:

```
return helper(0, 0, 0)
```

Is there a bug in this part of the function? Why or why not?

◯ Yes, because it's possible the starting cell has a skyscraper

◯ Yes, because it's possible the ending cell has a skyscraper

◯ No, because before we recurse we haven't seen any skyscrapers yet

◯ No, because the starting cell is always a house

**5. (8.0 points) Parts and Pairs and Words, Oh My!**

Given a list of `words`, we want to classify each word by its part of speech. (Understanding grammar isn't important for this question.)

We'll define *noun* as a word that ends in "er" ("er" is the *suffix*, the last two letters); a *verb* as a word that ends in "ed". If the word is neither a noun nor a verb, we will classify it as *unknown*. We want `parts_of_speech` to return a *count of the unknown words*, along with a list of "pairs", a tuple mapping each word to it's part of speech.

**Reminder:** Read the skeleton code carefully! For full credit, your solution needs to fit in the blanks. Each part will be graded independently; if `word_pair` doesn't work correctly, you can still earn points on the other parts.

```
def word_pair(word):
    """
    >>> word_pair("petted")
    ('petted', 'verb')
    """
    word_suffix = word[____(a)______]
    parts = { "ed": "verb", "er": "noun" }
    if ___(b)___ in parts:
        return (word, ____(c)____)
    return (word, "unknown")

def parts_of_speech(words):
    """
    >>> parts_of_speech(["pet"])
    (1, [('pet', 'unknown')])
    >>> parts_of_speech(["petter", "petted", "pets"])
    (1, [('petter', 'noun'), ('petted', 'verb'), ('pets', 'unknown')])
    """
    classifications = list( __(d)__ (____(e)____, words) )
    unknowns = list( filter(lambda pair: ____(f)____, classifications) )
    return len(unknowns), classifications
```

**(a) (1.0 pt)** Fill in blank (a). (For clarity, please include the `[ ]` in your response.)

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

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

**(d) (1.0 pt)** Which function completes blank (d)?

- ◯ `map`
- ◯ `filter`
- ◯ `reduce`
- ◯ None of the above

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

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

**(g) (1.0 pt)** Let's say we now have a large list of word pairs, and we want to *group* them by their part of speech, in a function `group_by_part`, such that we return a list of parts of speech paired to their list of words, e.g. `(part_of_speech, [list_of_words])`.

Consider a function which would be used like this:

```
>>> group_by_part([
...     ('petted', 'verb'), ('pets', 'unknown'), ('runner', 'noun'),
...     ('student', 'unkown'), ('studied', 'verb')
... ])
[ ('verb', [ 'petted', 'studied' ]), ('noun', [ 'runner' ]), ('unknown', [ 'pets', 'student' ]) ]
```

Of the following 3 functions, which would allow us to correctly inmplement `group_by_part`, assuming we could write a correct function to accomplish the task?

◯ `reduce`

◯ `filter`

◯ `map`

**6. (7.0 points) C88C Shorts**

You are tasked with creating a new short form video content app for C88C, since the last really popular app was shut down! However, the app has some constraints: The posted short must be **at most** 88 seconds long and contain at least one of the following topics: loops, hofs, lists, and lambdas.

Complete the `shorts` function, which takes in an integer `length` and returns a function `validate`.

`validate` takes in a `topics` list and returns four possible strings:

- `"Short is too long and does not contain required topic!"` if the video's length does not satisfy the condition and the short does not include a required topic.
- `"Short is too long!"` if the video does not satisfy the length condition, but satisfies the topic condition.
- `"Short does not contain a required topic!"` if the video does not satisfy the topic condition, but satisfied the length condiiton.
- `"Succesfully posted"` if the post meets both of the required constraints.

```
def shorts(length):
    """
    >>> validate_short_1 = shorts(88)
    >>> validate_short_1(['lambdas'])
    'Successfully posted'
    >>> validate_short_1(['recursion', 'loops'])
    'Successfully posted'
    >>> validate_short_2 = shorts(98)
    >>> validate_short_2(['lambdas'])
    'Short is too long!'
    >>> validate_short_3 = shorts(78)
    >>> validate_short_3(['recursion'])
    'Short does not contain a required topic!'
    >>> validate_short_4 = shorts(99)
    >>> validate_short_4(['recursion'])
    'Short is too long and does not contain required topic!'
    """
    valid_length = _____(a)_____
    def validate(topics):
        valid_topic = _____(b)_____
        required_topics = ['loops', 'hofs', 'lists', 'lambdas']
        for topic in topics:
            if topic in required_topics:
                valid_topic = _____(c)_____
                break
        if _____(d)_____:
            return "Short is too long and does not contain required topic!"
        elif _____(e)_____:
            return "Short is too long!"
        elif _____(f)_____:
            return "Short does not contain a required topic!"
        else:
            return "Successfully posted"
    return _____(g)_____
```

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

| |
|---|
| |

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

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

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

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

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

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

**7. (8.0 points) Stock Portfolio Analysis**

You've been asked to help analyze a portfolio of stocks, where a `portfolio` of stocks is represented as a list of dictionaries. (You don't need to know anything about stocks to solve this question, and the values in the dictionaries are just examples.)

Each stock dictionary has the following key-value pairs:

| Key | Value |
|---|---|
| `"stock"` | name of the stock (`str`) |
| `"quantity"` | quantity of the stock owned (`int`) |
| `"buy_price"` | the purchase price per share (`int`) |

**NOTE:** It is possible to have multiple dictionaries with the **same** stock name, which would represent different times you purchased a stock.

You want to analyze the portfolio to determine the total value of each company's shares. To calculate the total value, you need to lookup the current price from a dictionary of `prices`, mapping an individual share to its current price.

- `analyze_portfolio` returns a dict that maps *each stock* name to the *total* current value of the stock owned. This is calculated by multiplying the quantity by current price for each dictionary with the same stock name, and then summing those values.
- `compute_total_value` takes the result of `analyze_portfolio` and will sum all the values to get a single total across all stocks.

Refer to the doctests for examples.

```
def analyze_portfolio(portfolio, prices):
    """
    >>> prices = { "AAPL": 150, "TSLA": 100, "GOOG": 125 }
    >>> portfolio = [
      {"stock": "AAPL", "quantity": 10, "buy_price": 90},
      {"stock": "TSLA", "quantity": 5, "buy_price": 50},
      {"stock": "GOOG", "quantity": 2, "buy_price": 120},
      {"stock": "AAPL", "quantity": 5, "buy_price": 100}, # 2nd AAPL stock
      {"stock": "TSLA", "quantity": 2, "buy_price": 100}
    ]
    >>> analyze_portfolio(portfolio, prices)
    {'AAPL': 2250, 'TSLA': 700, 'GOOG': 250}
    """
    company_total = {}
    for stock_dict in portfolio:
      stock_name = stock_dict["stock"]
      current_price = ____(a)____
      if _____(b)_____ in company_total:
        company_total[stock_name] += _____(c)_____
      else:
        company_total[stock_name] = _____(c)_____

    return company_total

def compute_total_value(stock_totals):
    """
    >>> subtotals = {'AAPL': 2250, 'TSLA': 500, 'GOOG': 250}
    >>> compute_total_value(subtotals)
    3000
    """
    ---------------------------
```

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

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

**(c) (2.0 pt)** Fill in blank (c). (Blank c appears twice, and both have the same solution.)

**(d) (4.0 pt)** Complete the function using any technique you have learned so far. You do not need all the lines provided, but you may use all of them if needed.

(The following doctests are copied from earlier for convenience.)

```
def compute_total_value(stock_totals):
    """
    >>> subtotals = {'AAPL': 2250, 'TSLA': 500, 'GOOG': 250}
    >>> compute_total_value(subtotals)
    3000
    """
```

```
def compute_total_value(stock_totals):

    ------------------------------------------------------------------------

    ------------------------------------------------------------------------

    ------------------------------------------------------------------------

    ------------------------------------------------------------------------

    ------------------------------------------------------------------------
```

**8. (5.0 points) Recursive Possession**

Given a list of words, we will be constructing a possessive noun phrase! We will add the suffix `'s` to the end of each word, except for the last word in the list, to signify possession of the last word. If the `make_plural` boolean value is `True`, then we need to make the last word plural by adding an "s".

**Note:** You may assume `words` will always contain at least 1 word.

```
def convert_to_possessive(words, make_plural):
    """
    >>> convert_to_possessive(["turtle"], True)
    'turtles'
    >>> convert_to_possessive(["ramya", "cat"], True)
    "ramya's cats"
    >>> convert_to_possessive(["ramya", "roommate", "dog"], False)
    "ramya's roommate's dog"
    """
    if _____(a)_____:
        if make_plural:
            return _____(b)_____
        else:
            return _____(c)_____
    possessive_word = _____(d)_____ + "'s "
    return possessive_word + _____(e)_____
```

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

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

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

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

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

**9. (7.0 points) Sharks and Guppies**

James and Elam are trying to model the ocean ecosystem but due to poor funding they can only support a basic model of marine life with two animals: guppy fish and sharks. In this question, you will complete the classes to model the ecosystem.

**Note:** For this question, you can use functions defined in previous subparts in the current subpart you are working on. You may assume that the functions in previous subparts are defined correctly.

**(a) (1.0 pt)** Below is a basic implementation of the `Guppy` class. Guppies have the following instance attributes:

- `x` and `y` (`float`): the coordinates of the guppy in the ocean
- `energy` (`float`): the guppy's energy level

```
class Guppy():
    def __init__(self, x, y, energy):
        self.x = x
        self.y = y
        self.energy = energy

    # returns the distance from the Guppy to the specified (x, y) point
    def distance(self, x, y):
        return ((self.x - x) ** 2 + (self.y - y) ** 2) ** 0.5
```

Now, since sharks are guppies (I've been told this is up for debate) fill in the blanks to make the constructor of the `Shark` class that inherits from `Guppy`:

```
class Shark(Guppy):
    def __init__(self, x, y, energy):

        ____________________________________________________________
```

**(b) (3.0 pt)** Now that we have the `Shark` class we want to simulate what happens when they try to eat a `Guppy`.

For a `Shark` to eat a `Guppy` it must have the same or more energy than the `cost` to eat the `Guppy`. The `cost` is defined as the sum of: the distance between the shark to the guppy and the guppy's energy.

If this condition is met then the shark will:

i. Gain triple the guppy's energy AND
ii. Lose energy equivalent to the `cost`

Below, implement the `eat` method of the `Shark` class that takes in a `Guppy` instance called `fish`.

```
def eat(self, fish):

    cost = ____________________________ + ____________________________

    if ____________________________________________________:

        self.energy = ____________________________________________________
```

**(c) (3.0 pt)** We now want to model whale sharks. `WhaleShark` inherits from `Shark`.

`WhaleShark`s are unique in that they can't eat `Shark`s but can eat `Guppy`s! (Remember, this is because not all `Guppy`s are `Shark`s, but all `Shark`s are `Guppy`s.)

Implement `eat_guppies` which will take in `guppies`, a list of `Guppy` objects, and eats each of them if they are **NOT** a `Shark`. A `WhaleShark` can eat a `Guppy` if it meets the same conditions as mentioned in the previous subpart.

Hint: `type(obj)` will return the type of `obj`. For example:

```
>>> s = "hello"
>>> type(s) == str
True
```

```
class WhaleShark(Shark):
    # __init__ method hidden.  You can assume it works correctly.
    def eat_guppies(self, guppies):

        for ___________________________ in ___________________________:

            if  ___________________________________ != Shark:

                self.eat(___________________________)
```

**10. (5.0 points) Petsmart**

Someone from Petsmart SF has come to us asking for help in creating a digital catalogue of all the cats they have available for adoption. In this problem, we will be implementing a Petsmart cat catalogue as an ADT.

The Catalogue ADT is internally represented as a dictionary. Each dictionary will have a key representing unique cat breeds (e.g. siamese, ragdoll), with its value being a list of all cat names with that breed.

After this we will implement some code that will allow us to create our catalogue, add cats, and do other functions.

**Note:** For this question, you can use functions defined in previous subparts in the current subpart you are working on. You may assume that the functions in previous subparts are defined correctly.

```
# Catalogue ADT

def make_catalogue():
    return {}

def add_cats(catalogue, names, breeds):
    # Part (a)

def get_breed(catalogue, name):
    # Part (b)

def get_names(catalogue, breed):
    # Part (c)

def remove_cats(catalogue, names):
    # Part (d)
```

**Testcases:** For your convenience, we've provided all doctests below. The doctests are in the same order as the subparts, so feel free to read the subparts and then look back at the doctests. Or, feel free to read all the doctests at once.

```
>>> c1 = make_catalogue()
>>> add_cats(c1, ['Katsu', 'Waffles', 'Earl'], ['Rex', 'British Longhair', 'Rex'])
>>> c1
{'British Longhair': ['Waffles'], 'Rex': ['Katsu', 'Earl']}
>>> get_breed(c1, 'Waffles')
'British Longhair'
>>> get_names(c1, 'Rex')
['Katsu', 'Earl']
>>> remove_cats(c1, ['Waffles', 'Katsu'])
>>> c1
{'Rex': ['Earl']}
```

**(a) (1.5 pt)** Implement the function `add_cats`, which is part of the Catalogue ADT. Given a catalogue `catalogue`, a list of cat names `names`, and a list of their breeds `breeds`, the function should add them into the catalogue. Assume that `len(names) == len(breeds)`.

```
def add_cats(catalogue, names, breeds):
    for i in range(len(names)):

        if ____________________________________________________________:

            catalogue[breeds[i]].append(_____________________________________)
        else:

            catalogue[breeds[i]] = ________________________________________
```

**(b) (1.5 pt)** Implement the function `get_breed`, which is part of the Catalogue ADT. Given a catalogue `catalogue` and a cat's name `name`, `get_breed` returns its breed. Assume that `name` exists in the catalogue and is unique.

```
def get_breed(catalogue, name):

    for breed in ____________________________________________________:

        if ____________________________ in ____________________________:
            return breed
```

**(c) (0.5 pt)** Implement the function `get_names`, which is part of the Catalogue ADT. It returns all cats' names of a certain breed in the given catalogue. Assume that `breed` exists in the catalogue.

```
def get_names(catalogue, breed):

    return ________________________________________________
```

**(d) (1.5 pt)** We also need to create a function if cats are adopted from Petsmart. Implement the function `remove_cats` (part of the Catalogue ADT) which takes in a catalogue and a list of names of cats to be adopted, and removes the cats from the catalogue. Assume that all names exist in the catalogue and are unique.

```
def remove_cats(catalogue, names):
    for name in names:

        breed = ____________________________________________________
        if catalogue[breed] == [name]:

            ____________________________________________________
        else:

            ____________________________________________________
```

**No more questions.**