Data C88C
Fall 2024

Almeda, DeNero, Tsang
Midterm

Print Your Name: ____

Print Your Student ID: ______________________________

Print Your @berkeley.edu email address: ______________________________

You have 110 minutes. There are 6 questions of varying credit (100 points total).

Question:	1	2	3	4	5	6	Total
Points:	15	32	15	23	15	0	100

For questions with **circular bubbles**,
you may select only one choice.

○ Unselected option (completely unfilled)

● Only one selected option (completely filled)

✓ Don't do this (it will be graded as incorrect)

- The exam is closed book, closed notes, closed computer, closed calculator, except two 8.5" x 11" pages of your own creation and the provided midterm study guide.
- Anything you write outside the answer boxes or you ~~cross out~~ will not be graded.
- If you write multiple answers, your answer is ambiguous, or the bubble is not entirely filled in, we will grade the worst interpretation.
- You may use built-in Python functions that do not require import, such as **`pow`**, **`len`**, **`abs`**, **`bool`**, **`int`**, **`float`**, **`str`**, **`round`**, **`max`**, **`min`**, **`list`**, **`tuple`**, **`sum`**, **`all`**, **`any`**, **`map`**, **`filter`**, **`zip`**, **`sorted`**, and **`reversed`**.
- You may **not** use example functions defined on your study guide unless a problem clearly states you can.
- You may not use ; to place two statements on the same line.
- You may use the `Link` class defined on the midterm study guide.

Sign (or type) your name to confirm that all work on this exam will be your own. The penalty for academic misconduct on an exam is an F in the course.

Sign (or print) your name: ____

Q1 *Into the Haunted House!* (15 points)

Dhruv and Andria stumble into a haunted house, and want to do some sleuthing to figure out the mysteries inside!

```python
flashlight = lambda item: print(item)
in_the_corner = "A patch of grass."

def haunted_house():
    print("Spooky!" and "Scary!")
    on_the_chandelier = lambda: "Look, it's Kenny!"
    on_the_windowsill = print
    flashlight(on_the_windowsill(on_the_chandelier()))


def peek(in_the_corner, where):
    flashlight(where())

def attic():
    in_the_corner = "A metal key."
    def basement():
        return in_the_corner
    return basement
```

Into the unknown!

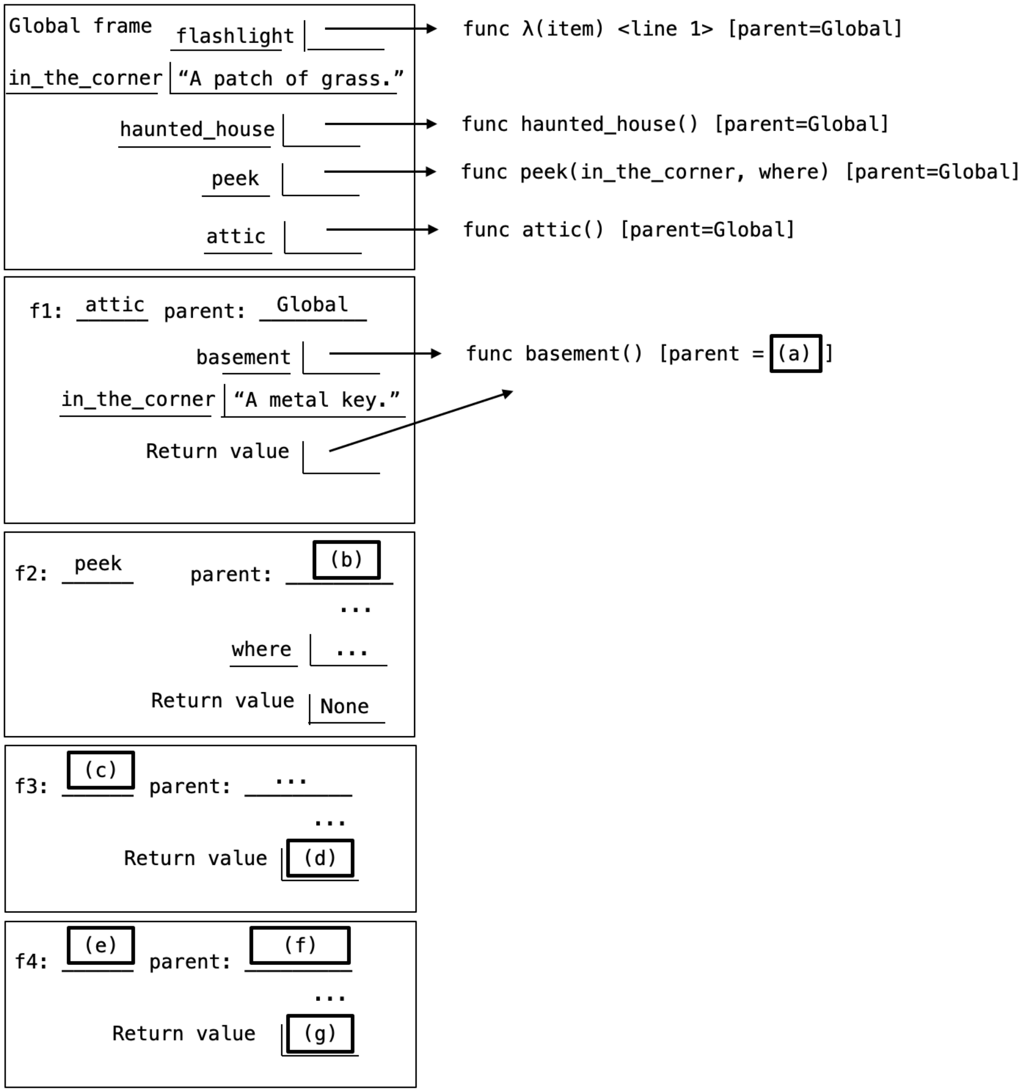
Q1.1 (5 points) Let's figure out what secrets we've uncovered. Note what the following call expression prints out.

If any of the lines errors, write "Error" and do not execute any further lines of code.

`haunted_house()`

Assume we do not call `haunted_house`. Using the code from the previous page, answer the following questions about the environment diagram that results from the following call expression:

`peek("cobwebs.", attic())`



Q1.2 (1.5 points) What should go in blank (a)?

○ Global ○ attic

○ f1

Q1.3 (1.5 points) What should go in blank (b)?

○ Global ○ attic

○ f1

Q1.4 (1.5 points) What should go in blank (c)?

○ attic ○ λ<line 1>

○ basement ○ peek

○ flashlight ○ where

Q1.5 (2 points) What should go in blank (d)?

○ "A patch of grass." ○ "cobwebs."

○ "A metal key." ○ None

Q1.6 (1 point) What should go in blank (e)?

○ attic ○ λ<line 1>

○ basement ○ peek

○ flashlight ○ where

Q1.7 (1 point) What should go in blank (f)?

○ Global ○ f3

○ f1 ○ attic

○ f2 ○ peek

Q1.8 (1.5 points) What should go in blank (g)?

○ "A patch of grass." ○ "cobwebs."

○ "A metal key." ○ None

Q2 *Ghostbusters!* (32 points)

A ”spooky list” is one that has a ghost hidden somewhere in it. Ghosts are represented by some spooky sequence. However, our ghosts have gotten crafty! As long as the spooky numbers appear in consecutive order, even within a nested list, our list is considered spooky.

Oski decides to take a stab at implementing `is_spooky`, which detects whether our list is spooky or not.

```
def is_spooky(s, sequence):
    """Returns whether s is a spooky list given a non-empty sequence.
    If the numbers in the sequence appear consecutively in s, return True.
    Otherwise, return False.
    >>> is_spooky([1, 2, 3], [1, 2, 3])
    True
    >>> is_spooky([[1], [2], [3]], [1, 2, 3])
    True
    >>> is_spooky([1, [2, [[[3]]]]], [1, 2, 3])
    True
    >>> is_spooky([1, [], 2, 3], [1, 2, 3])
    True
    >>> is_spooky([0, [1, 2], 3, 0], [1, 2, 3])
    True
    >>> is_spooky([], [1, 2, 3])
    False
    >>> is_spooky([1, 2, 4, 3], [1, 2, 3])
    False
    >>> is_spooky([123], [1, 2, 3]) #Spooky numbers should not be combined!
    False
    """
    return sequence in s   #OSKI'S BUGGY IMPLEMENTATION
```

Q2.1 (3 points) Give one example input s for a sequence [8, 0, 0] where Oski’s implementation would successfully detect a spooky list.

Q2.2 (3 points) Give one example input s for a sequence [8, 0, 0] where Oski’s implementation would **fail** to detect a spooky list. In other words, `is_spooky` would return True when the spooky list is not spooky, or would return False when the spooky list is indeed spooky.

(8 points) In order to help us detect these ghosts, fill in the following implementation of **flatten**, which takes in a potentially nested list and returns a flattened list. A flattened list contains only integers as elements within the list (i.e. there should not be lists within lists in the returned list).

```
def flatten(s):
    """Returns a flattened version of s.
    >>> flatten([])
    []
    >>> flatten([1, 2, 3])
    [1, 2, 3]
    >>> flatten([1, 2, [[[3]]]])
    [1, 2, 3]
    """
        flat_list = []

        for __________________________________________________:
                                  Q2.3
            if type(elem) == list:

                ____________________________________________________________
                                          Q2.4
            else:

                ____________________________________________________________
                                          Q2.5
        return flat_list
```

Fill in the following implementation of `how_spooky`, which takes in a list `s` and a non-empty list `sequence`. It returns the number of ghosts inside of `s`. Assume `flatten` is correctly implemented.

```
def how_spooky(s, sequence):
    """Returns the number of ghosts inside of list s.
    Ghosts take on the values in sequence.
    >>> how_spooky([], [1])
    0
    >>> how_spooky([8, 1, 0, 0], [8, 0, 0])
    0
    >>> how_spooky([8, 0, 0], [8, 0, 0])
    1
    >>> how_spooky([1, [8], [], [0, [[0]]]], [8, 0, 0])
    1
    >>> how_spooky([0, 0, 0], [0, 0])
    2
    >>> how_spooky([1, 2, 3, 1, 2, 3, 1, 2, 3], [1, 2])
    3
    >>> how_spooky([1, [2], [[3, 1], 2, 3], 1, 2, 3], [1, 2])
    3
    """
    flattened_list = BLANK ONE
    def how_spooky_helper(lst):
        if BLANK TWO:
            return 0
        elif BLANK THREE:
            return BLANK FOUR
        else:
            return BLANK FIVE
    return how_spooky_helper(flattened_list)
```

Q2.6 (1 point) What should go in `BLANK ONE`?

○ `flatten(s)`

○ `s.flatten()`

○ `flatten()`

Q2.7 (3 points) What should go in `BLANK TWO`?

○ `len(lst) < len(sequence)`

○ `len(lst) != len(sequence)`

○ `len(lst) == len(sequence)`

○ `len(lst) > len(sequence)`

○ `lst[0] != sequence[0]`

○ `lst != sequence`

○ `lst[1:] != sequence[1:]`

Q2.8 (3 points) What should go in `BLANK THREE`?

○ `lst == sequence`

○ `lst[0] == sequence[0]`

○ `lst[1:] == sequence[1:]`

○ `lst[:len(sequence)] == sequence`

○ `any([lst[x] == sequence[x] for x in range(len(sequence))])`

Q2.9 (3 points) What should go in `BLANK FOUR`?

○ `how_spooky_helper(lst)`

○ `how_spooky_helper(lst[1:])`

○ `how_spooky_helper(lst[:len(lst) - 1])`

○ `how_spooky_helper(lst[len(sequence):])`

○ `1 + how_spooky_helper(lst)`

○ `1 + how_spooky_helper(lst[1:])`

○ `1 + how_spooky_helper(lst[:len(lst) - 1])`

○ `1 + how_spooky_helper(lst[len(sequence):])`

Q2.10 (3 points) What should go in `BLANK FIVE`?

○ `how_spooky_helper(lst)`

○ `how_spooky_helper(lst[1:])`

○ `how_spooky_helper(lst[:len(lst) - 1])`

○ `how_spooky_helper(lst[len(sequence):])`

○ `1 + how_spooky_helper(lst)`

○ `1 + how_spooky_helper(lst[1:])`

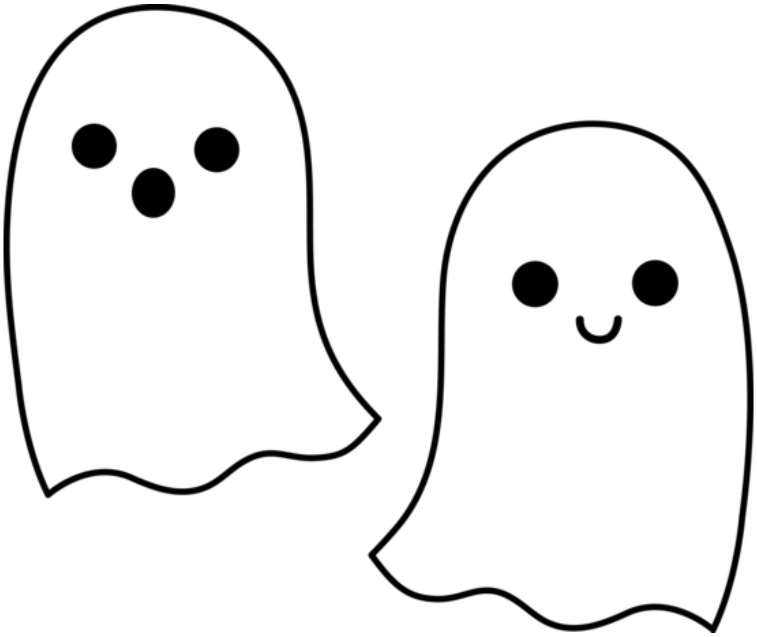
○ `1 + how_spooky_helper(lst[:len(lst) - 1])`

○ `1 + how_spooky_helper(lst[len(sequence):])`

(5 points) Fill in the following implementation of `scariest_list`. Given a list of potentially spooky lists and a sequence, `scariest_list` returns the list with the most ghosts inside of it. You may assume that all code from previous subparts have been implemented correctly. In the case of a tie, return the the list that comes first in spooky_lists. **You may not use any square brackets in your answer (i.e. neither [or] should appear in your answer).**

```
def scariest_list(spooky_lists, sequence):
    """Returns the list with the most ghosts.
    >>> scariest_list([[], [1, 2], [1, 1, 1, 2]], [1])
    [1, 1, 1, 2]
    >>> scariest_list([[1, 2], [2, 2, 2], [3, 4]], [2])
    [2, 2, 2]
    >>> scariest_list([[1, 2], [1, [2, [1, [2]]]]], [1, 2])
    [1, [2, [1, [2]]]]
    >>> scariest_list([[1, 2, 1], [1, 3, 1], [1, 1]], [1])
    [1, 2, 1]
    """

    return ____________________________________________________________
                                   Q2.11
```



Spotting the ghosts :O

Q3 *Trick or Treat!* **(15 points)**

The C88C staffers decide to go trick-or-treating! Suppose we have the following code that helps us determine what each staffer gets:

```
def trick_or_treat(staffer):
    reward = "Nothing"
    if len(staffer) % 2 == 0:
        reward = "Trick"
    if len(staffer) // 2 >= 3:
        reward = "Chocolate!"
    elif staffer[0] == "S":
        reward = "Cookies!"
    return reward
```

Select what each of the following expressions evaluates to:

Q3.1 (1.5 points) `trick_or_treat("Swetha")`

○ "Nothing"

○ "Trick"

○ "Chocolate!"

○ "Cookies!"

○ None

Q3.2 (1.5 points) `trick_or_treat("Jedi")`

○ "Nothing"

○ "Trick"

○ "Chocolate!"

○ "Cookies!"

○ None

Q3.3 (2 points) `trick_or_treat(["Shm", "and", "Grace"])`

○ "Nothing"

○ "Trick"

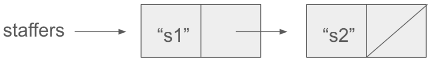
○ "Chocolate!"

○ "Cookies!"

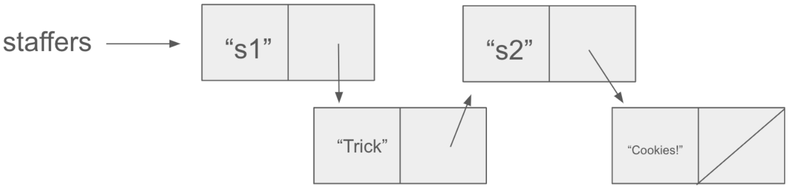
○ None

Q3.4 (10 points) Implement **candy_bag**, which takes in a Linked List called **staffers** and modifies **staffers** such that every staff member's **rest** attribute is another **Link** instance containing the name of that staffer's reward (determined by passing their name into **trick_or_treat**. If **staffers** is an empty Linked List, we should not modify anything.

For instance, if staffer **"s1"** gets "Trick" and **"s2"** gets "Cookies!", the linked list:



turns into this linked list after calling **candy_bag** on it:



```
def candy_bag(staffers):
    """Modifies staffers.
    Each staffer becomes linked to their respective trick/treat.
    >>> staffers = Link("s1", Link("s2"))
    >>> candy_bag(staffers)
    >>> staffers.first
    's1'
    >>> staffers.rest.first #Assume "s1"'s reward is "Trick"
    'Trick'
    >>> staffers.rest.rest.first
    's2'
    >>> staffers.rest.rest.rest.first #Assume "s2"'s reward is "Cookies!"
    'Cookies!'
    """

    current = ________________________________________
                             Q3.5

    while ________________________________________:
                         Q3.6

        reward = trick_or_treat(____________________________________)
                                               Q3.7

        reward_link = Link(__________________, __________________)
                                   Q3.8                 Q3.9

        ___________________________ = reward_link
                    Q3.10

        current = ____________________________________
                                  Q3.11
```

Q4 *Pumpkin Patch Pyramid* **(23 points)**

Angela and Khadija are wandering around the pumpkin patch, and want to make pyramids of pumpkins as a fun C88C staff activity. In order for the pumpkin patch pyramid to be symmetrical (with a pumpkin aligned exactly in the middle), each layer must only have an odd number of pumpkins in it. They wonder what all the combinations of pumpkins there could be given n pumpkins!

Implement `sums`, which takes a positive integer `n` and returns a list of all unique combinations of odd numbers that sum to `n`.

```
def sums(n):
    """List all the lists of unique odd numbers that sum to n.
    >>> sorted(sums(3))
    [[3]]
    >>> sorted(sums(16))
    [[1, 3, 5, 7], [1, 15], [3, 13], [5, 11], [7, 9]]
    >>> sorted(sums(17))
    [[1, 3, 13], [1, 5, 11], [1, 7, 9], [3, 5, 9], [17]]
    """
    def at_least(n, k):
        if n < k:
            return BLANK ONE
        elif n == k:
            return BLANK TWO
        with_k = [BLANK THREE for s in BLANK FOUR]
        without_k = BLANK FIVE
        return BLANK SIX
    return at_least(n, BLANK SEVEN)
```

This is not a pumpkin patch pyramid, but it is cute...

(Question 4 continued...)

Q4.1 (2 points) What should go in `BLANK ONE`?

○ `0`

○ `[]`

○ `[[]]`

○ `[0]`

Q4.2 (3 points) What should go in `BLANK TWO`?

○ `[n]`

○ `[k]`

○ `[[n]]`

○ `[[k]]`

Q4.3 (4 points) What should go in `BLANK THREE`?

○ `[k] + s`

○ `[s] + k`

○ `s`

○ `[s, k]`

Q4.4 (4 points) What should go in `BLANK FOUR`?

○ `sums(n - 2)`

○ `sums(n - 1)`

○ `at_least(n, k - 2)`

○ `at_least(n, k - 1)`

○ `at_least(n, k)`

○ `at_least(n, k + 1)`

○ `at_least(n, k + 2)`

○ `at_least(n-k, k-2)`

○ `at_least(n-k, k-1)`

○ `at_least(n-k, k)`

○ `at_least(n-k, k+1)`

○ `at_least(n-k, k+2)`

Q4.5 (4 points) What should go in `BLANK FIVE`?

○ `sums(n - 2)`

○ `sums(n - 1)`

○ `at_least(n, k - 2)`

○ `at_least(n, k - 1)`

○ `at_least(n, k)`

○ `at_least(n, k + 1)`

○ `at_least(n, k + 2)`

○ `at_least(n-k, k-2)`

○ `at_least(n-k, k-1)`

○ `at_least(n-k, k)`

○ `at_least(n-k, k+1)`

○ `at_least(n-k, k+2)`

Q4.6 (3 points) What should go in `BLANK SIX`?

○ `with_k + without_k`

○ `with_k.extend(without_k)`

○ `with_k - without_k`

○ `[lst for lst in with_k if lst not in without_k]`

○ `[lst for lst in with_k if lst in without_k]`

○ `max([with_k, without_k], key = len)`

Q4.7 (3 points) What should go in `BLANK SEVEN`?

○ `k`

○ `n`

○ `0`

○ `1`

Q5 *Spooky Scary Skeletons* (15 points)

Jack-O-Lantern wants to track his roster of Skeletons!

Fill in the `Skeleton` and `Spooky Scary Skeleton` classes with the following specifications:

The `Skeleton` class contains a class attribute `roster`, which is a dictionary with names (Strings) as keys and `Skeleton` or `Spooky Scary Skeleton` instances as values. Every time a new `Skeleton` or `Spooky Scary Skeleton` is created, we modify this class attribute.

Each `Skeleton` and `Spooky Scary Skeleton` has the following instance attributes:

- name: a String representing the name of the Skeleton
- dance_move: a String representing the Skeleton's best dance move!

Additionally, implement the `dance` method, which takes in a String representing the name of a potential dance `partner`.

- If this name is not in the `roster`, return the String "Skeleton Stranger Danger!"
- Else, if both `Skeleton's` `dance_move`s are equal to one another, return the String "Dancing the night away!"
- Else, return the String "Not quite in sync!"

One caveat: if both Skeletons are `Spooky Scary Skeleton`s, then we should return "Fated to be!", regardless of their favorite `dance_move`.

Below is an example use case of the classes:

```
>>> skellington = Skeleton("Skellington", "Bone Boogie")
>>> grimm = SpookyScarySkeleton("Grimm", "Bone Boogie")
>>> jack = SpookyScarySkeleton("Jack", "Rib Cage Rumba")
>>> skellington.name
'Skellington'
>>> Skeleton.roster["Grimm"].dance_move
'Bone Boogie'
>>> skellington.dance("Jack")
'Not quite in sync!'
>>> skellington.dance("Grimm")
'Dancing the night away!'
>>> jack.dance("Grimm")
'Fated to be!'
>>> jack.dance("Priya")
'Skeleton Stranger Danger!'
```

```
class Skeleton:
    roster = {}
    def __init__(self, name, dance_move):

        ______________________________ = ______________________________
                     Q5.1                              Q5.2

        ______________________________ = ______________________________
                     Q5.3                              Q5.4

        ____________________________________________________________
                                   Q5.5
    def dance(self, partner):

        if _________________________________________________________________:
                                          Q5.6
            return "Skeleton Stranger Danger!"

        elif ____________________________________________________________:
                                          Q5.7
            return "Dancing the night away!"
        else:
            return "Not quite in sync!"

class SpookyScarySkeleton(Skeleton):
    def dance(self, partner):
        if CODE OMITTED: #Is the partner a Spooky Scary Skeleton?
            return "Fated to be!"

        return ____________________________________________________________
                                          Q5.8
```

(5 points) Jack finds himself dancing with other `Skeleton`s quite often. Fill in the method implementation for `get_dance_method` such that the following code executes as described below. For full credit, you may **not** use lambdas in your solution. A correct implementation with lambdas will incur a point penalty.

```
... #Assume we have the code from the previous page
>>> jack_dance = get_dance_method(jack)
>>> jack_dance("Grimm")
'Fated to be!'
>>> jack_dance("Priya")
'Skeleton Stranger Danger!'
```

```
class Skeleton:
    ...
    def get_dance_method(skeleton):

        return __________________________________________________
                                   Q5.9
```

Q6 *The Finish Line* (0 points)

These questions will not be assigned credit; feel free to leave them blank.

Q6.1 (0 points)

○ Trick ○ Treat

Q6.2 (0 points) What's your favorite halloween candy?

Q6.3 (0 points) If there's anything else you want us to know, or you feel like there was an ambiguity in the exam, please put it in the box below.

For ambiguities, you must qualify your answer and provide an answer for both interpretations. For example, "if the question is asking about A, then my answer is X, but if the question is asking about B, then my answer is Y". You will only receive credit if it is a genuine ambiguity and both of your answers are correct. We will only look at ambiguities if you request a regrade.