DATA C88C
Fall 2025

Michael Ball
Final

Your Name: _____

Your Student ID: _____

Exam Room: _____

Student ID of the person to your left: _____

Student ID of the person to your right: _____

You have 180 minutes. There are 10 questions of varying credit. (110 points total)

Question:	1	2	3	4	5	6	7	8	9	10	Total
Points:	18	12	12	10	12	12	10	10	14	0	110

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

- ○ Unselected option (Completely unfilled)
- ◌ Don't do this (it will be graded as incorrect)
- ● Only one selected option (completely filled)

For questions with **square check boxes**, you may select one or more choices.

- ■ You can select
- ■ multiple squares
- ☑ (Don't do this)

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/checkbox is not entirely filled in, we will grade the worst interpretation. For coding questions with blanks, you may write at most one statement per blank and you may not use more blanks than provided.

As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others. I will follow the rules of this exam.

Acknowledge that you have read and agree to the honor code above and sign your name below:

_____

SID: _____

Q1 *Potpourri of Python* 🐍 **(18 points)**

For each of these questions, assume that the code is executed in a fresh Python interpreter, unless otherwise specified. If any of the code results in an error, select "Error" as your answer (for multi-choice questions) or write "Error" (for free-response questions).

Q1.1 (1 point) Assume the following code is executed.

```
>>> siblings = ['pj', 'teddy', 'toby', 'charlie', 'gabe']
>>> duncan = ['bobs', 'bugs', 'be', 'gone']
>>> siblings.append(siblings.extend([ w for w in duncan if len(w) % 2 == 0 ]))
>>> len(siblings)
```

What is the output of the last line?

- ◯ 5
- ◯ 9
- ◯ 7
- ◯ 10
- ◯ 8
- ◯ `Error`

Q1.2 (1 point) Assume the following code is executed. (The first two lists are the same as the previous question.)

```
>>> siblings = ['pj', 'teddy', 'toby', 'charlie', 'gabe']
>>> duncan = ['bobs', 'bugs', 'be', 'gone']
>>> list(map(lambda x: len(siblings[x]) + len(duncan[x]), range(len(duncan))))
```

What is the output of the last line?

- ◯ `[6, 9, 6, 11]`
- ◯ `[2, 5, 4, 7]`
- ◯ `[6, 7, 4, 9]`
- ◯ `Error`
- ◯ `[6, 9, 8, 11]`

Q1.3 (1 point) Assume the following code is executed. (The first two lists are the same as the previous questions.)

```
>>> siblings = ['pj', 'teddy', 'toby', 'charlie', 'gabe']
>>> duncan = ['bobs', 'bugs', 'be', 'gone']
>>> list(map(lambda x: len(siblings[x]) + len(duncan[x]), range(len(siblings))))
```

What is the output of the last line? (*Yes, the last line is different from before.*)

- ◯ `[6, 9, 6, 11, 8]`
- ◯ `[2, 5, 4, 7, Error]`
- ◯ `[6, 7, 4, 9, Error]`
- ◯ `Error`
- ◯ `[6, 9, 6, 11, 0]`

Q1.4 (1 point) Assume the following code is executed, for parts 1.4 to 1.6. Recall that `string.split()` splits a string into a list of words, separated by a space. e.g., `'hello world'.split()` returns `['hello', 'world']`.

```
food = [ 'siu mai', 'cha siu bao', 'egg tart', 'pineapple bun',
   'cheung fun', 'ha gau', 'xiao long bao' ]
i = 0
item = None
while i < len(food):
    parts = food[i].split()
    if len(parts) == 3:
        item = food[i]
    i += len(parts)
```

What is the value of `item` after executing the above code?

- ◯ `'siu mai'`
- ◯ `'cheung fun'`
- ◯ `'cha siu bao'`
- ◯ `'ha gau'`
- ◯ `'egg tart'`
- ◯ `'xiao long bao'`
- ◯ `'pineapple bun'`
- ◯ `Error`

Q1.5 (2 points) Continuing from the previous question, what is the final value of `i` after executing the above code?

Q1.6 (2 points) Which of the following modifications to the above code would cause it to result in an infinite loop? Consider each answer option independently; that is, only one change is made at a time.

- ☐ Changing `i += len(parts)` to `i += 1`
- ☐ Changing `if len(parts) == 3:` to `if len(parts) >= 1:`
- ☐ Changing `i += len(parts)` to `i -= 1`
- ☐ Changing `while i < len(food):` to `while i > len(food):`
- ☐ Adding the line `food.extend(food)` right after the line `i += len(parts)`
- ◯ None of the above

Q1.7 (1 point) Assume the following code is executed:

```
>>> m = map(lambda x: x * x, range(2, 3))
>>> m
<map object at 0x10534e5f0>
>>> next(m)
```

What is the output of the last line?

- ◯ `0`
- ◯ `4`
- ◯ `6`
- ◯ `9`
- ◯ `Error`

Q1.8 (1 point) Continuing from the previous question, what is the output of the next line, an additional call to `next(m)`?

```
>>> next(m)
```

What is the output of the last line?

- ◯ `0`
- ◯ `4`
- ◯ `6`
- ◯ `8`
- ◯ `9`
- ◯ `16`
- ◯ `Error`

Q1.9 (2 points) Given that we can call `next()` on the result of a `map` or `filter` function, what type of object is returned by `map` and `filter` in Python 3?

- ◯ `list`
- ◯ `set`
- ◯ `iterator`
- ◯ `dictionary`
- ◯ `generator function`
- ◯ `Error`

Q1.10 (2 points) Considering the `WHERE` statement in SQL, which of the following Python functions is most similar in functionality to `WHERE`?

- ◯ `map`
- ◯ `filter`
- ◯ `reduce`
- ◯ `zip`
- ◯ `range`
- ◯ `Error`
- ◯ None of the above

Q1.11 (1 point) In lecture, we saw a function called `reverse`, which correctly reverses a string. A slightly modified version of the function is shown below (and is also correct). However, it has an unexpected behavior when called with a list argument.

```
def reverse(s):
    if not s:
        return ''
    return reverse(s[1:]) + s[:1]
```

What happens when we call `reverse([1, 2, 3])`?

- ◯ `[1, 2, 3]`
- ◯ `[3, 2, 1]`
- ◯ `123`
- ◯ `'321'`
- ◯ `Error`

Q1.12 (2 points) Considering the same `reverse` function from the previous question, which of the following modifications would allow it to correctly reverse both strings and lists?

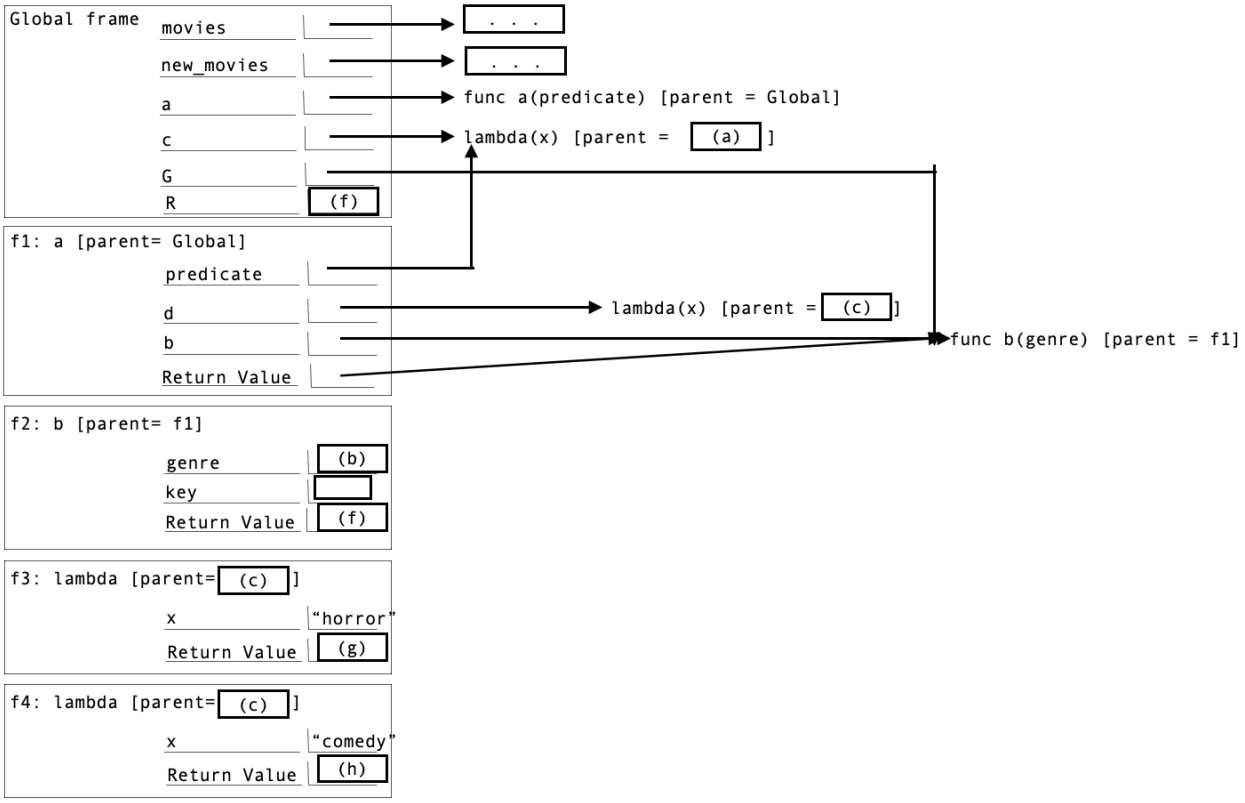
- ◯ Change the base case to return `s` instead of `''`
- ◯ Change the condition in the base case to `if len(s) == 0:` instead of `if not s:`
- ◯ Change the recursive case to use `s[0]` instead of `s[:1]`
- ◯ Change the recursive case to use `s.pop()` instead of `s[:1]`
- ◯ Change the recursive case to use `s[-1]` instead of `s[:1]`
- ◯ Use `append` instead of the `+` operator
- ◯ It is not possible to modify the function to handle both types

Q1.13 (1 point) What is the runtime of `reverse`, given the input `n` is the length of the string or list being reversed?

- ◯ $O(1)$
- ◯ $O(\log(n))$
- ◯ $O(n)$
- ◯ $O(n^2)$
- ◯ $O(2^n)$

Q2 *An Inconvenient Truth: Environment Diagrams* 🌳 **(12 points)**

Fill in the blanks to complete the environment diagram. Assume code has been fully run before filling in blanks. The blanks with no labels have no questions associated with them and are not scored. Please note: If more than one blank shares the same label (e.g., '(c)'), they have the same answer.




```
1  def a(predicate):
2      d = lambda x: x[0] == "c"
3      def b(genre):
4          for key in movies:
5              if d(key):
6                  movies[key].append(new_movies[genre][0])
7              else:
8                  movies[key].pop()
9          return "movies"
10     return b
11
12 movies = {"horror": ["IT", "Conjuring"], "comedy": ["Elf", "Home Alone"]}
13 new_movies = {"horror": ["Annabelle", "Until Dawn"], "comedy": ["Hangover", "Ted"]}
14
15 c = lambda x: print("value")
16
17 G = a(c)
18 R = G("comedy")
```

Q2.1 (1 point) Fill in blank (a).

Q2.2 (1 point) Fill in blank (b).

Q2.3 (1 point) Fill in blank (c).

Q2.4 (2 points) After the last successful iteration, what is the value of `movies["horror"]`?

- ◯ `["IT", "Conjuring"]`
- ◯ `["IT"]`
- ◯ `["Conjuring"]`
- ◯ `["IT", "Conjuring", "Annabelle"]`
- ◯ `[]`

Q2.5 (2 points) After the last successful iteration, what is the value of `movies["comedy"]`?

Q2.6 (2 points) Fill in blank (f).

Q2.7 (1.5 points) Fill in blank (g).

Q2.8 (1.5 points) Fill in blank (h).

Q3 *Christmas Gifts* 🎁 **(12 points)**

Sarah wants to figure out where to buy a Christmas gift. Implement a higher order function:

- The outer function, `find_gift`, takes in a string `gift` representing the item Sarah is looking for and returns the inner function.
- The inner function, `search`, takes in an arbitrarily nested dictionary `gift_dict` representing the locations of gifts and returns a string representing the path to the desired `gift` from the outer function (see the doctests for examples).

The `gift_dict` has the following structure:

1. **Key**: The location of a gift (string)
2. **Value**: Either:
    1. The name of an item in that location (string), or
    2. Another dictionary structured the same way as `gift_dict`

See the doctests for examples of what the `gift_dict` could look like. For example, `gift_dict3` is a dictionary where the gift "MacBook Pro" is located in the "Macs" department of the "Apple Store" on "4th Street". You may assume `gift_dict` will never be empty and the gift will always appear exactly once in `gift_dict`, either directly or inside a nested dictionary.

Hint: The built-in function `isinstance` checks if an item is of a certain data type. For example, `isinstance([1, 2, 3], list)` returns `True`.

```
>>> gift_dict1 = {"Target": "socks", "Nordstrom": "lipstick"}
>>> find_gift("socks")(gift_dict1)
'Target'

>>> gift_dict2 = {
...     "Target": "socks",
...     "Nordstrom": {"Makeup": "lipstick", "Clothes": "shirt"}
... }
>>> find_gift("shirt")(gift_dict2)
'Nordstrom -> Clothes'

>>> gift_dict3 = {
...     "4th Street": {
...         "Apple Store": {"Phones": "iPhone 14" , "Macs": "MacBook Pro"}
...     },
...     "Target": "socks",
...     "Nordstrom": { "Makeup": "lipstick", "Clothes": "shirt" }
... }
>>> find_gift("MacBook Pro")(gift_dict3)
'4th Street -> Apple Store -> Macs'
>>> find_gift("socks")(gift_dict3)
'Target'
>>> find_gift("lipstick")(gift_dict3)
'Nordstrom -> Makeup'
```

```
def find_gift(gift):
    def search(gift_dict):
        for store in gift_dict:
            current_gift = ___(a)___
            if ___(b)___:
                return store
            if ___(c)___:
                location = ___(d)___
                if location:
                    return f"{__(e.1)__} -> {__(e.2)__}"
    return ___(f)___
```

Q3.1 (2 points) Fill in blank (a).

Q3.2 (2 points) Fill in blank (b).

Q3.3 (2 points) Fill in blank (c).

Q3.4 (2 points) Fill in blank (d).

Q3.5 (2 points) Fill in blanks (e.1) and (e.2):

```
return f"{__________________________} -> {__________________________}"
```

Q3.6 (2 points) Fill in blank (f).

Q4 *A Model for Learning* 🤖 **(10 points)**

Polly recently got a job interview for Ngoogle and needs to review some machine learning concepts. To do so, she created a class, `Model`, to help her prep. Read through the doctests and provided code to answer the following questions.

Note: Use the `finished_learning` method to check if Polly has more facts to learn for a topic (she only needs to learn 3 facts for each), instead of hard coding this functionality yourself.

```
class Model:
  """
  >>> lin_reg = Model("Linear Regression")
  >>> lin_reg.num_facts_learned
  0
  >>> lin_reg.finished_learning()
  False

  >>> for _ in range(3):
  ...   lin_reg.learn()
  Fact: Linear Regression predicts a continuous numeric value.
  Fact: It assumes a roughly linear relationship between features and the target.
  Fact: A common loss function is Mean Squared Error.
  >>> lin_reg.num_facts_learned
  3

  >>> lin_reg.learn()
  No more learning! Polly mastered Linear Regression.
  >>> lin_reg.finished_learning()
  True
  >>> lin_reg.learn()
  No more learning! Polly mastered Linear Regression.

  >>> dec_trees = Model("Decision Trees")
  >>> dec_trees.num_facts_learned
  0
  >>> dec_trees.learn()
  Fact: A Decision Tree predicts by following a path of feature-based questions.
  >>> dec_trees.num_facts_learned
  1
  """
  def __init__(self, name):
    self.name = name
    self.num_facts_learned = ___(a)___
    # This is an iterator that returns 3 facts (strings) one at a time
    self.fact_iterator = # you do not need to know how this is implemented

  def finished_learning(self):
    return self.num_facts_learned == 3
```

```
def learn(self):
  """
  Polly learns the next fact, if any remain.
  """
  if ___(b)___:
    print(f"No more learning! Polly mastered {___(c)___}.")
  else:
    print(f"Fact: {___(d)___}")
    ___(e)___
```

Q4.1 (1 point) Fill in blank (a).

Q4.2 (1 point) Fill in blank (b).

Q4.3 (1 point) Fill in blank (c).

Q4.4 (2 points) Fill in blank (d).

Q4.5 (1 point) Fill in blank (e).

When Polly tries to learn about neural networks her brain feels fried! Implement the `NeuralNetwork` class that inherits from `Model` and overrides the `learn` method.

Note: Your implementation must **avoid repetition** as much as possible.

```
class NeuralNetwork(Model):
  """
  >>> neural_net = NeuralNetwork("Neural Networks")

  >>> neural_net.learn()
  Fact: Neural Networks use layers of connected neurons.

  >>> neural_net.learn()
  Fact: Neural Networks can learn very complex patterns in data.

  >>> neural_net.learn()
  Fact: Neurons apply weights, bias, and an activation.
  Polly's brain feels fried.

  >>> neural_net.learn()
  No more learning! Polly mastered Neural Networks.
  Polly's brain feels fried.
  """
  def learn(self):
    ___(f)___
    ___(g)___:
        print(___(h)___)
```

Q4.6 (2 points) Fill in blank (f).

Q4.7 (1 point) Fill in blank (g).

Q4.8 (1 point) Fill in blank (h).

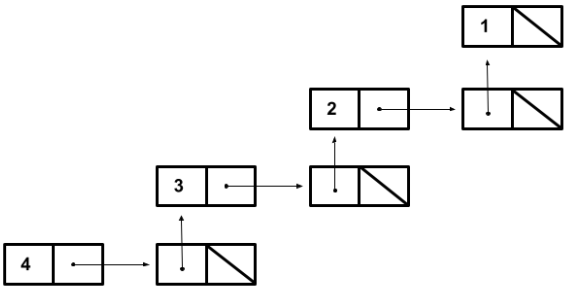
Q5 *Zigzag Linked Lists* 🌀 **(12 points)**

Your task is to implement the function `number_to_zigzag`, which takes in a non-negative integer `num` and **returns** a zigzagged Linked List with digits of `num` encoded within it in **reverse order**. In a zigzagged Linked List, each digit of the number is stored in a staggered structure: every node's `rest` attribute points to another `Link` whose `first` attribute holds the next node in the sequence (see the diagram).

Number

1234

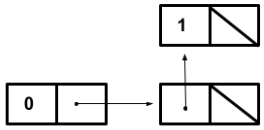
Zigzag Linked List



Number

10

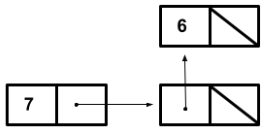
Zigzag Linked List



Number

67

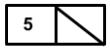
Zigzag Linked List



Number

5

Zigzag Linked List



```
def number_to_zigzag(num):
  """Takes a non-negative integer num and returns a zigzagged linked list
  of its digits in reverse order.
  >>> number_to_zigzag(1234)
  Link(4, Link(Link(3, Link(Link(2, Link(Link(1)))))))
  >>> number_to_zigzag(67)
  Link(7, Link(Link(6)))
  >>> number_to_zigzag(5)
  Link(5)
  >>> number_to_zigzag(10)
  Link(0, Link(Link(1)))
  """
  if ___(a)___:
    return Link(num)
  last = ___(b)___
  smaller = number_to_zigzag(___(c)___)
  return Link(last, ___(d)___)
```

Q5.1 (1 point) Select all options that could fill in blank (a).

- ☐ `num < 10`
- ☐ `num <= 10`
- ☐ `num < 0`
- ☐ `num <= 0`
- ☐ `num == 0`

Q5.2 (1 point) Fill in blank (b).

Q5.3 (1 point) Fill in blank (c).

Q5.4 (1 point) Fill in blank (d).

Next, implement the function **sum_zigzag**, which takes in a zigzagged Linked List and returns the sum of all digits stored in it.

```
def sum_zigzag(zigzag):
  """Takes a zigzagged linked list and returns the sum of its digits.
  >>> zig = number_to_zigzag(12034)
  >>> sum_zigzag(zig)
  10
  >>> zag = number_to_zigzag(1)
  >>> sum_zigzag(zag)
  1
  """
  if ___(e)___:
    return 0
  if ___(f)___:
    return zigzag.first
  else:
    return ___(g)___
```

Q5.5 (3 points) Select all options that could fill in blank (e).

Hint: Be sure to check the reference sheets for the definition of the `Link` class.

- ☐ `zigzag is Link.empty`
- ☐ `zigzag is not Link.empty`
- ☐ `not zigzag`
- ☐ `len(zigzag) == 0`
- ☐ `not isinstance(zigzag, Link)`
- ☐ `isinstance(zigzag, Link)`

Q5.6 (2 points) Fill in blank (f).

- ◯ `zigzag == Link.empty`
- ◯ `zigzag.first is Link.empty`
- ◯ `zigzag.rest is Link.empty`
- ◯ `zigzag.rest is not Link.empty`
- ◯ `isinstance(zigzag.rest, Link)`

Q5.7 (3 points) Fill in blank (g).

SID: _____

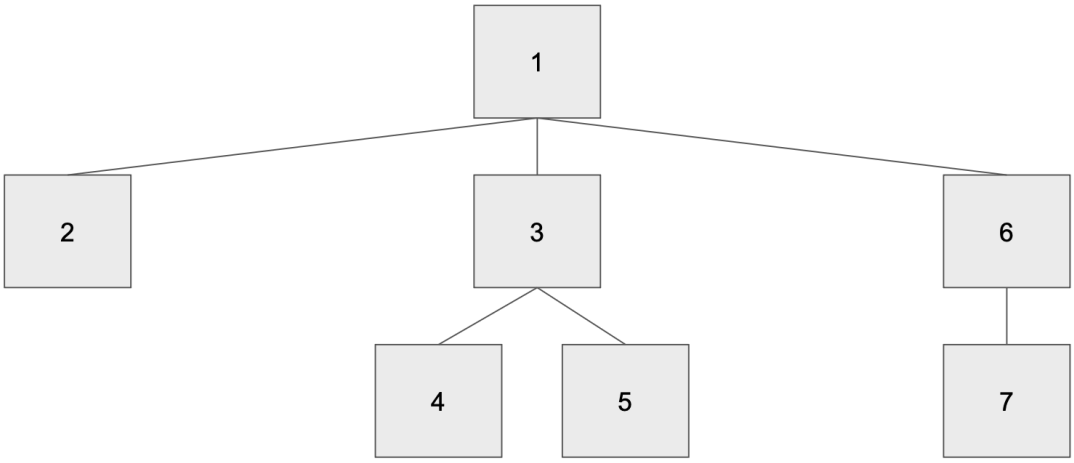
Q6 *Sadly, These Aren't Christmas Trees* 🎄 **(12 points)**

Grace's favorite number is `k`, and she wants to count how many nodes in a tree `t` which have exactly `k` branches.

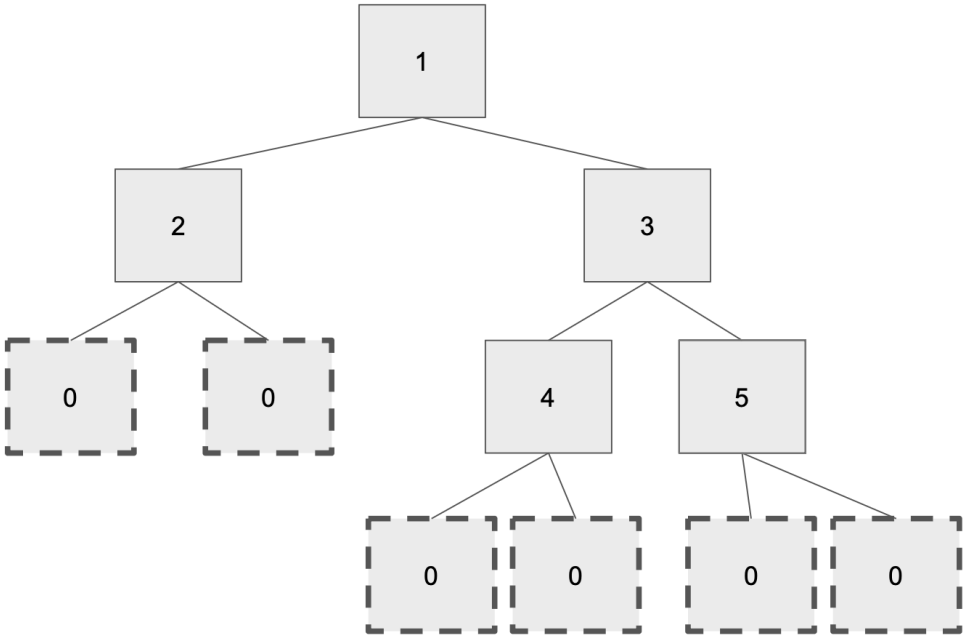
Maryam, however, wants to add her favorite number `v` to the tree while also ensuring that every node has exactly `k` branches. Implement `count_add_k(t, k, v)` which does both of the following:

1. **Returns** the number of nodes in the **original tree** `t` that have exactly `k` branches
2. **Mutates** `t` so that:
   - Any node with more than `k` branches keeps only its first `k` branches.
   - Any node with fewer than `k` branches adds leaves with value `v` until it has exactly `k` branches.

For example, with `k = 2`, the tree (`t`) below has *one* node with exactly k branches (the node with label `3`).



After applying `count_add_k(t, 2, 0)`, `1` is returned, and the tree is mutated to the `Tree` below. The nodes with dashed borders were added, and the branch containing the nodes `6` and `7` was removed.



```
def count_add_k(t, k, v):
    """
    Return how many nodes in the original tree t that have exactly k branches.
    Also mutate t so that every node has exactly k branches.

    Note that t is the same tree shown in the diagram on the previous page.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)]), Tree(6, [Tree(7)])])
    >>> count_add_k(t, 2, 0)
    1
    >>> t
    Tree(1, [
        Tree(2, [ Tree(0), Tree(0) ]),
        Tree(3, [
            Tree(4, [ Tree(0), Tree(0) ]),
            Tree(5, [ Tree(0), Tree(0) ])
          ]
        )
      ]
    )
    """
    if ___(a)___:
        count = 1
    else:
        count = 0

    for b in ___(b)___:
        count += ___(c)___

    if len(t.branches) > k:
        t.branches = ___(d)___
    while ___(e)___:
        ___(f)___
    return count
```

Q6.1 (2 points) Fill in blank (a).

Q6.2 (1 point) Fill in blank (b).

Q6.3 (2 points) Fill in blank (c).

Q6.4 (2 points) Select all options that could fill in blank (d).

- ☐ `t.branches[:k]`
- ☐ `t.branches[k:]`
- ☐ `t.branches[:]`
- ☐ `[t[i] for i in range(k)]`
- ☐ `[t.branches[i] for i in range(k)]`

Q6.5 (2 points) Fill in blank (e).

Q6.6 (2 points) Fill in blank (f).

Q6.7 (1 point) What is the runtime of `count_add_k`, given the input `n` is the number of nodes in the tree? (Assume that the value of `k` is negligible compared to `n`, e.g. treat `k` like a small constant.)

- ○ $O(1)$
- ○ $O(\log(n))$
- ○ $O(n)$
- ○ $O(n^2)$
- ○ $O(2^n)$

Q7 *LinkIterator* **(10 points)**

In C88C, the implementation of the `Link` class is not iterable. To allow us to iterate through linked lists easily, you decide to make our `Link` class iterable!

Below is the updated `Link` class. It now has an `__iter__` method, which returns a `LinkIterator` object.

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __iter__(self):
        return LinkIterator(self)
```

Implement the `LinkIterator` class, so that you can use a for loop to iterate over all elements of a linked list without manually accessing `first` and `rest`.

```
class LinkIterator:
  """
  >>> lnk = Link(1, Link(2, Link(3)))
  >>> for lnk in lnk:
  ...     print(lnk)
  1
  2
  3
  """
    def __init__(self, link):
        self.current = link

    def __iter__(self):
        ___(a)___

    def __next__(self):
        if ___(b)___:
            raise StopIteration
        value = ___(c)___
        self.current = ___(d)___
        ___(e)___
```

Q7.1 (1 point) Fill in blank (a).

- ◯ `yield self`
- ◯ `return self`
- ◯ `return iter(self)`

Q7.2 (2 points) Fill in blank (b).

Q7.3 (1 point) Fill in blank (c).

Q7.4 (1 point) Fill in blank (d).

Q7.5 (1 point) Fill in blank (e).

Q7.6 (2 points) Assuming the `Link` class has been updated correctly, given the linked list: `lnk = Link(1, Link(2, Link(3)))`, select all following statements that are true.

- ☐ `lnk` is an iterator
- ☐ `lnk` is an iterable
- ☐ `iter(lnk)` returns an iterator object that can be used with `next`
- ☐ You can call `next(lnk)` to get the first element of `lnk`

Q7.7 (2 points) Assume that `LinkIterator` has been implemented correctly. Select the sequence of function calls that will output 1 then 3.

```
1 lnk = Link(1, Link(2, Link(3)))
2 def odd_gen(lnk):
3     for elem in lnk:
4         if elem % 2 != 0:
5             yield elem
```

○
```
gen = odd_gen(lnk)
next(gen)
next(gen)
```

○
```
odd_gen(lnk)
next(odd_gen)
next(odd_gen)
```

○
```
iter = iter(lnk)
next(iter)
next(iter)
```

○ None of the above

Q8 *Pythonic SQL* **(10 points)**

Recall that SQL is a *declarative* programming language, which means that the programmer describes the output they want rather than a series of steps to achieve that output. Rebecca has implemented some basic SQL functionality in Python, but she needs your help to fix a bug!

The `Row` class (which has **no bugs**) represents the data in a single row of a `Table`. For example:

```
>>> columns = ['english_word', 'spanish_word']
>>> values = ['hello', 'hola']
>>> row = Row(columns, values)
>>> row['english_word']
'hello'
>>> row['spanish_word']
'hola'
```

The `Table` class represents a SQL table made up of 1 or more rows. Any code for this class that is not necessary for the problem has been omitted and you should assume it works properly.

The `__init__` method (which has **no bugs**) takes in a list of column names called `columns` and a 2D list of `row_values` to create a `Table` instance:

```
1  def __init__(self, columns: list[str], row_values: list):
2    """
3    >>> columns = ['actor', 'character', 'tv_show']
4    >>> row_values = [
5    ...     ['Ella Purnell', 'Jinx', 'Arcane'],
6    ...     ['Ella Purnell', 'Lucy MacLean', 'Fallout'],
7    ...     ['Hailee Steinfeld', 'Vi', 'Arcane'],
8    ...     ['Hailee Steinfeld', 'Kate Bishop', 'Hawkeye'],
9    ...     ['Pedro Pascal', 'Joel Miller', 'The Last of Us'],
10   ...     ['Caleb McLaughlin', 'Lucas Sinclair', 'Stranger Things'],
11   ... ]
12   >>> characters = Table(columns, row_values)
13   """
14   self.columns = columns
15   self.rows = [ Row(columns, values) for values in row_values ]
```

The `select` method (which has **no bugs**) takes in a list of column names called `columns` and returns a new `Table` which is equivalent to performing a `SELECT ... FROM ...` statement in SQL. For example, `characters.select(['actor', 'character'])` is equivalent to `SELECT actor, character FROM characters;`

The **buggy** `select_where` method takes in a list of column names and a one-argument function `predicate` which takes in a `Row` object and returns `True` if that row should be included in the output and `False` otherwise. This is equivalent to performing a `SELECT ... FROM ... WHERE ...` statement in SQL.

```
def select_where(self, columns: list[str], predicate):
  """
  Doctest below is equivalent to:
  SELECT actor, character FROM characters WHERE actor = 'Ella Purnell';

  >>> characters.select_where(
  ...   ['actor', 'character'],
  ...   lambda row: row['actor'] == 'Ella Purnell'
  ... )
  actor | character
  ----------------
  Ella Purnell | Jinx
  Ella Purnell | Lucy MacLean
  """
  selected_table = self.select(columns)
  filtered_row_values = []
  for row in selected_table.rows:
      if predicate(row):
          filtered_row_values.append([row[col] for col in columns])
  filtered_table = Table(columns, filtered_row_values)
  return filtered_table
```

Q8.1 (2 points) Suppose we are working with the `characters` table defined above. What is the equivalent SQL statement for:

```
characters.select_where(['character'], lambda row: 'Pedro' in row['actor'])
```

◯ `SELECT character FROM characters WHERE actor = 'Pedro';`

◯ `SELECT character FROM characters WHERE actor LIKE '%Pedro%';`

◯ `SELECT character FROM characters WHERE actor LIKE 'Pedro%';`

◯ `SELECT character FROM characters WHERE 'Pedro' IN actor;`

Q8.2 (2 points) Suppose we are working with the `characters` table defined above. Which of the following **will cause an error** with the buggy implementation of `Table`? Select all that apply.

☐
```
characters.select_where(
  ['actor'],
  lambda row: 'Pedro' in row['actor']
)
```

☐
```
characters.select_where(
  ['actor', 'tv_show'],
  lambda row: 'Caleb' in row['actor'] or 'Hawkeye' == row['tv_show']
)
```

☐
```
characters.select_where(
  ['actor', 'tv_show'],
  lambda row: len(row['character']) > 4
)
```

○ None of the above

Q8.3 (4 points) Describe the bug in the `select_where` method in 1-2 sentences. You do not need to propose a fix nor do you need to provide line numbers, unless it helps clarify your explanation.

Q8.4 (2 points) Consider just this part of the buggy **`select_where`** method:

```
filtered_row_values = []
for row in selected_table.rows:
    if predicate(row):
        filtered_row_values.append([row[col] for col in columns])
```

Let c be the number of columns in the table and r be the number of rows in the table. Assume that the **`predicate`** function and **`append`** method each take $O(1)$ time. What is the (worst-case) runtime of the code snippet above?

- ○ $O(1)$
- ○ $O(\log(c))$
- ○ $O(\log(r))$
- ○ $O(c)$
- ○ $O(r)$
- ○ $O(c * r)$
- ○ $O(c^2)$
- ○ $O(r^2)$

Q9 *Apartment Hunting* 🏠 **(14 points)**

It's the start of apartment hunting season at Berkeley, and members of C88C staff are struggling to find apartment-mates for next year. Dhruv and Reema want to use SQL to identify everyone's preferences. The tables in use are described below and all of their data is shown below.

`staff`: contains a unique `id` (string) for every person, their `name` (string), preferred `location` (string), whether they want a `single` or double room (boolean, `TRUE` if they want a single or `FALSE` if they want a double), `budget` (integer), and `in_state` status (boolean, `TRUE` if they are an in-state student or `FALSE` if they are an out-of-state student).

id	name	location	single	budget	in_state
SD57	Dhruv	Downtown	FALSE	2000	FALSE
SA13	Reema	North	TRUE	1150	TRUE
DM51	Mike Baller	San Francisco	TRUE	3000	TRUE
SA17	Cynthia	South	FALSE	1000	TRUE
GA13	Mira	South	TRUE	1200	TRUE
YS20	Alicia	Downtown	TRUE	1100	FALSE
BD28	Grace B	South	TRUE	1300	TRUE
BK28	Orazaly	North	TRUE	1100	FALSE
PV12	Thompson	South	FALSE	1300	FALSE
SM21	Isabelle	South	FALSE	1000	TRUE
SG53	Maryam	North	FALSE	2000	TRUE
KH39	Grace X	Downtown	FALSE	1900	FALSE
PM31	Rebecca	Downtown	FALSE	2500	TRUE

`apartments`: records each potential housing option with columns `rent` (integer), `location` (string), option for `single` room (`TRUE` means they have single rooms and `FALSE` means they have double rooms), and `distance`, in miles, to campus (float).

name	rent	location	single	distance
Dwight	2000	Downtown	FALSE	0.9
Identity	900	Downtown	TRUE	0.8
Standard	1400	South	FALSE	0.2
Modera	1300	North	FALSE	0.6
Hillside	1200	North	TRUE	0.7
Den	1500	South	FALSE	0.3
Edgewater	2500	San Francisco	FALSE	13.9
NEMA	2800	San Francisco	TRUE	11.7
Blake	1200	Downtown	TRUE	1.0
Panoramic	1100	South	FALSE	0.5
Hearst	800	North	TRUE	0.6
Berk	1000	South	TRUE	0.5

Q9.1 (2 points) Which column in the `staff` table would preserve the original number of rows when used in a `GROUP BY` clause? Select all that apply and consider each answer option independently, e.g. there is only 1 column in each hypothetical `GROUP BY` clause.

- ☐ `id`
- ☐ `location`
- ☐ `budget`
- ☐ `name`
- ☐ `single`
- ☐ `in_state`

Q9.2 (1 point) Select the statement to complete the query below to correctly return all apartments that contain the letter "a" anywhere in their name.

```
1 SELECT * FROM apartments WHERE _____;
```

- ◯ `name LIKE '%a%'`
- ◯ `name LIKE '_a_'`
- ◯ `name LIKE '%a'`
- ◯ `name LIKE '_a'`
- ◯ `name LIKE 'a%'`
- ◯ `name LIKE 'a_'`

Q9.3 - Q9.9 (8 points) Dhruv and Reema want to write a SQL query to identify all staff who have a potential apartment option, given the following criteria:

1. The `staff` member's `location` and the `apartment`'s `location` are the same.
2. The `staff` member's room preference (`single`) and the `apartment`'s room options (`single`) are the same.
3. The `apartment`'s `rent` must be less than or equal to the `staff` member's `budget`.

Your query should return the following columns: `staff_name`, `in_state`, `apartment_name`, `budget_surplus` (calculated as the difference between a staff member's `budget` and the apartment's `rent`). The `staff_name` column should be in alphabetical order, and, for each person, the `budget_surplus` is ordered from largest to smallest.

The resulting table should look like the one below.

staff_name	in_state	apartment_name	budget_surplus
Alicia	FALSE	Identity	200
Dhruv	FALSE	Dwight	0
Grace B	TRUE	Berk	300
Grace X	FALSE	Identity	1000
Grace X	FALSE	Blake	700
Maryam	TRUE	Modera	700
Mike Baller	TRUE	NEMA	200
Mira	TRUE	Berk	200
Orazaly	FALSE	Hearst	300
Rebecca	TRUE	Dwight	500
Reema	TRUE	Hearst	350
Thompson	FALSE	Panoramic	200

```
SELECT
  s.name AS staff_name,
  s.in_state AS in_state,
  a.name AS apartment_name,

  _____ AS budget_surplus
      Q9.3
FROM staff AS s

JOIN _____ AS a
          Q9.4

ON _____ AND _____
      Q9.5          Q9.6

WHERE _____
           Q9.7

ORDER BY _____ ASC, _____ DESC;
          Q9.8          Q9.9
```

Assume the previous query is correct and the output table is stored as **`preferred`**.

The evil landlord twins, Arnav and Arvind, decide to increase everyone's rent! With this new policy, each staffer's **`budget_surplus`** should now be reduced to 98% of its original value. Return a table that computes the average budget surplus for in-state and out-of-state staffers after the rent increase.

The expected output is shown below.

in_state	average_budget_surplus
FALSE	392
TRUE	367.5

```
SELECT in_state, ____(a)____(____(b)____) AS average_budget_surplus
FROM preferred
GROUP BY in_state;
```

Q9.10 (1 point) Fill in blank (a).

Q9.11 (2 points) Fill in blank (b).

Q10 *Just for fun!* **(0 points)**

Q10.1 Draw something fun, or write a message for the staff!