Spring 2023 In-class Midterm Exam Overview

On Friday, April 21st, you will take a midterm exam during your CS111 lecture.

Your exam will be hand-written by you on paper. Each question has boxes where you will be asked to write your answer. Only write inside the boxes. We will scan your exam into Gradescope and will grade what you write within the boxes only (so do not write in the margins or outside the boxes).

In the exam you could be asked to:

1. Read Python programs and explain what they do. What values do they print or return?
2. Modify existing Python programs.
3. Write Python programs that satisfy a specification.

The exam is open notes in the sense that you can bring with you any **printed and handwritten materials**, such as your written notes, printouts of slides and web pages you think are important, and books. We strongly recommend against printing large numbers of pages, since most students don't have time during the exam to consult them. To prepare for the exam, it's better for you to write a few pages of your own notes of what you think is important and might forget.

You are **not** allowed to use any devices during the exam, including but not limited to computers, calculators or smartphones. You are **not** allowed to browse the web during the exam nor use a Python interpreter during the exam.

Here are some things we encourage you to do to prepare for the exam:

- **Practice solving lots of problems involving Python concepts and coding.** Where can you find such problems?
  - This document has several problems from past midterms and quizzes
  - Problems we didn’t get to in the lecture notebooks.
  - Problems you didn’t get to in lab
  - Problems in the slides
  - Redo problems from quizzes and psets.
- Review the quizzes and corresponding solutions
- Review the posted solutions for all psets. Often, the posted solutions may show you how to solve a problem in a better way than you did on your pset.
- Review all course lecture slides, notebooks and lab materials. Write down anything you're confused about and ask an instructor/tutor.

Concepts for Midterm Exam

The exam covers all material in the course up to and including Lec 19 (CSV Format and Real World Data), Lab 11 (Dictionaries II), and Project 8.

Topics covered on the exam include the following:

- Multiple assignment (assignment with tuples on the left hand side of =). For example: `a, b = (4, 8)`.
- Nested loops
- List comprehension and operations such as mapping and filtering.
- Memory Diagrams
  - Memory Diagrams show the state of a program involving variables and values, including mutable values like lists and objects.
  - They are especially important for understanding:
    - assignment: changing the value stored in a variable, list slot, or object instance variable.
    - adding or removing slots in a list.
    - aliasing: the same mutable value can be accessed by multiple paths.
    - **==** vs. **is** for testing value equality.
  - Be able to translate from a given memory diagram to python code that produces such a diagram and also, given python code, draw the corresponding memory diagram to reflect the state of the variables and values.
- Sorting using `sorted()` and `.sort()`
- Debugging
  - Test cases
  - Tracing
  - Counterexamples
- Dictionaries:
  - Map keys to values.
  - Use subscripting notation, [...], to get and set value for key.
  - `.keys()`, `.values()`, `.items()`, `.get()` methods are helpful
  - common use is to keep the frequency count of items.

- Advanced combination of these mutable data structures: dictionaries of dictionaries, list of dictionaries, dictionaries of lists. Conversion from one data structure to another.
- Files:
  - Handling various files types (text, csv, json)
  - `csv.DictReader, csv.DictWriter`
  - `json.dump` and `json.load`
  - Operations for reading and writing files include `open`, `.readlines`, `.readline`, `.read`, and `.write`.
    - The special `with … as` statement for file manipulation will implicitly close files upon completion.
- Exceptions:
  - used for exceptional situations -- e.g., opening a file that does not exist, dividing by zero.
  - handled by `try/except`.

Problems on the midterm might also involve topics covered by the first midterm, including:

- Python expressions and statements
- Variables and assignments.
- Simple data types and operations on these: numbers (integer and floats), strings, booleans, `None`.
- Functions:
    - understanding the difference between function definition and function invocation.
    - function parameters
        - The name of parameters do not matter as long as they are used consistently.
        - In a function invocation frame, each parameter denotes a local variable initialized to the argument value.
    - understanding the difference between `return` and `print`.
    - The invocation frame model explains function calls.
- Scope:
    - the locality of parameters and other local variables assigned within a function body.
    - the `global` declaration for declaring a variable global within a function.
- Conditional statements: `if` statements with optional `elif` and `else` clauses.
- Sequences:
    - lists are mutable sequences of values. You can both change indexed slots and add and remove indexed slots from a list.
    - tuples are immutable sequences of values.
    - strings are immutable sequences of characters.
- Iteration:
    - Iterations are repeated updates to state variables, as expressed in iteration tables via iteration rules
    - Iterations are expressed in Python using loops:
        - `while` loops
        - `for` loops range over lists and are just `while` loops in disguise
        - loop gotchas:
            - premature return from the function (or early return)
            - updates to state variables in wrong order
        - Sometimes you **want** to return early from a loop via `return` or `break`
        - nested loops
    - Iterations can also be expressed with list comprehensions, which are a compact notation for mapping, filter, and appending patterns.
- Writing and using functions.
- Abstraction:
    - Python functions abstract over behavior.
- Problem solving strategies:
    - Divide/conquer/glue (or Divide/Solve/Combine)
    - Designing iterations (loops) with iteration tables and iteration rules
    - Incremental programming: work towards a full solution by starting with a simple solution that does very little adding more features and detail

Review Problems

The remainder of this handout includes some problems to help guide your review for the exam. These review problems do not necessarily reflect the difficulty of exam problems (these review problems are in many cases harder than ones you might encounter on the exam), but they cover concepts that you are expected to know for the exam. These problems are not exhaustive; your exam may include problems with topics different from those below.

If the problem asks for the output or result, you should try to *predict* this using pencil and paper, because you won't have a Python interpreter during the exam. Check your prediction using a Python interpreter.

Problem 0: Interpreting Functions

Show what is printed by the following program:

```
def chat():
      print("meow")
      return "bark"

def petTalk(num):
      if num < 5:
         chat()
         return "chirp"
      return chat()

print(petTalk(0)+petTalk(10))
```

Problem 1: Nested loops

You're trying to determine how hard it is to get from city to city by air travel. A "flight list" is defined as a list of strings, where the list represents a single airline's flights, and each string represents a flight, with the contents of the string indicating whether the flight is direct (`'D'`) or connecting (`'C'`). For example, this airline has 3 direct and 2 connecting flights:

['D', 'D', 'C', 'C', 'D']

Write a function `countDirectFlights` which takes a list of flight lists and returns the total number of direct flights across all the flight lists given. For example:

```
In [1]: data = [['D', 'D', 'C', 'C', 'D'], ['C', 'C', 'C'], ['D']]

In [2]: countDirectFlights(data)
Out[2]: 4
```

should compute the number 4, since there are 4 direct flights across the entire dataset.

Problem 2: Reading and Writing From Files

Suppose that calendar events are stored in a file format illustrated by the the contents of the following file `events.txt`, which contains four events from the Wellesley College calendar:

2022-04-18-Patriot's day
2022-04-26-Ruhlman Conference
2022-05-06-Last day of classes
2022-05-27-Commencement

Each event has four fields that are separated by the dash character `'-'`: (1) a four-digit year; (2) a two-digit month number; (3) a two-digit date; and (4) an event name. You may assume that the event name does not contain a dash.

2.1 Reading event tuples

Define a function `readEvents` that takes the name of a file with the format of `events.txt` and **returns** a **list of event tuples** where there is one event tuple for each line in the file. Each event tuple should be a tuple with four **strings** (no numbers!):

- a year string (like `'2022'`)
- a month string (like `'05'`)
- a day string (like `'06'`)
- an event name (like `'Last day of classes'`)

Here is an example of how your function should work:

```
>>> readEvents('events.txt')
[('2022', '04', '18', "Patriot's day"),
 ('2022', '04', '26', 'Ruhlman Conference'),
 ('2022', '05', '06', 'Last day of classes'),
 ('2022', '05', '27', 'Commencement')
]
```

Define the `readEvents` function. Your definition should use the `open` function to open the file for reading, and loop over the lines of the file to accumulate a list of event tuples. It is also recommended that you:

- use the `.strip` function to remove the newline at the end of a line;
- use the `.split` function to split a line on the dash character;
- use the `tuple` function to convert a list to a tuple.

2.2 Writing event tuples

Define a function `writeEvents` that takes (1) a filename and (2) a list of events tuples (like those returned by `readEvents` above), and writes each event tuple to one line of the file in the dash-separated format shown above.

For example, suppose that `myEventTuples` is defined as

```
myEventTuples = [
 ('2022', '04', '18', "Patriot's day"),
 ('2022', '04', '26', 'Ruhlman Conference'),
 ('2022', '05', '06', 'Last day of classes'),
 ('2022', '05', '27', 'Commencement')
]
```

Then `writeEvents('myEvents.txt', myEventTuples)` should cause the `myEvents.txt` file to have the following contents:

```
2022-04-18-Patriot's day
2022-04-26-Ruhlman Conference
2022-05-06-Last day of classes
2022-05-27-Commencement
```

Define the `writeEvents` function. Your definition should use the `open` function to open the file for writing, and loop over the event tuples to write a line for each tuple into the file (using the file `.write` method). Don't forget to add an explicit newline character `'\n'` at the end of each line. It is recommended (but not required) to use Python's f-strings to simplify writing to files.

Problem 3: Working with dictionaries and lists [short example]

Part a) You are given a list of dictionaries, where each dictionary has three keys: `'author'`, `'book'`, `'year'`, for example:

```
{'author': 'Jane Austen', 'book': 'Persuasion', 'year': 1818}
```

The list might have many such dictionaries, each with information about a specific book from a specific author. An example of the list is shown below:

```
authors = [{'author': 'Jane Austen', 'book': 'Persuasion', 'year': 1818},
{'author': 'Virginia Woolf', 'book': 'The Voyage Out', 'year': 1915},
{'author': 'Jane Austen', 'book': 'Emma', 'year': 1815}, {'author':
'Virginia Woolf', 'book': 'Mrs Dalloway', 'year': 1925}, {'author':
'Virginia Woolf', 'book': 'Orlando', 'year': 1928}, {'author': 'Jane
Austen', 'book': 'Pride and Prejudice', 'year': 1813}, {'author': 'Virginia
Woolf', 'book': 'Night and Day', 'year': 1919}, {'author': 'Jane Austen',
'book': 'Sense and Sensibility', 'year': 1811}, {'author': 'Virginia
Woolf', 'book': 'To the Lighthouse', 'year': 1927}, {'author': 'Jane
Austen', 'book': 'Northanger Abbey', 'year': 1818}, {'author': 'Virginia
```

```
Woolf', 'book': 'The Waves', 'year': 1931}, {'author': 'Jane Austen',
'book': 'Mansfield Park', 'year': 1814}]
```

Define the function `booksSortedByDate` which takes as a parameter the list of all dictionaries and the name of an author and returns a string that combines together the information of books about the author in a certain format. For example:

```
print(booksSortedByDate(authors, "Jane Austen"))
```

will display:

```
Jane Austen: Sense and Sensibility, 1811
Jane Austen: Pride and Prejudice, 1813
Jane Austen: Mansfield Park, 1814
Jane Austen: Emma, 1815
Jane Austen: Persuasion, 1818
Jane Austen: Northanger Abbey, 1818
```

And:

```
print(booksSortedByDate(authors, "Virginia Woolf"))
```

Will display:

```
Virginia Woolf: The Voyage Out, 1915
Virginia Woolf: Night and Day, 1919
Virginia Woolf: Mrs Dalloway, 1925
Virginia Woolf: To the Lighthouse, 1927
Virginia Woolf: Orlando, 1928
Virginia Woolf: The Waves, 1931
```

Part b) We would like to remove some repetitive information from our list of dictionaries, thus, we will convert it from a list to a dictionary, where the keys are the names of the authors. Concretely the list from Part a) after the conversion looks like the following; notice how the field `'author'` has disappeared:

```
authorsDict = {'Jane Austen': [
                    {'book': 'Persuasion', 'year': 1818},
                    {'book': 'Emma', 'year': 1815},
                    {'book': 'Pride and Prejudice', 'year': 1813},
                    {'book': 'Sense and Sensibility', 'year': 1811},
                    {'book': 'Northanger Abbey', 'year': 1818},
                    {'book': 'Mansfield Park', 'year': 1814}],
              'Virginia Woolf': [
                    {'book': 'The Voyage Out', 'year': 1915},
                    {'book': 'Mrs Dalloway', 'year': 1925},
                     {'book': 'Orlando', 'year': 1928},
```

```
                    {'book': 'Night and Day', 'year': 1919},
                    {'book': 'To the Lighthouse', 'year': 1927},
                    {'book': 'The Waves', 'year': 1931}]}
```

Define the function `groupByAuthor` that, given a list of dictionaries as in Part a), returns a dictionary like `authorsDict`.

Hint: The method `.pop()` for dictionaries will make the solution shorter.

Problem 4: Working with CSV files and dictionaries

For this problem, you are provide the file `grammy-winners.csv` in the `finalExamReview`
Folder (you can access this in the cs111 download folder), which contains the names of 50 artists with the most Grammy wins, according to this Wikipedia list.
Below are shown the few first lines of the CSV file.

```
nominee,sex,wins,nominations
Quincy Jones,M,28,80
Beyoncé,F,28,79
Alison Krauss,F,27,42
Stevie Wonder,M,25,74
```

Part 1: Get the artists from the file

Using the module csv and its function DictReader, you will write a function, **`getGrammyWinners`,** to read the content of the CSV file as a list of simple dictionaries (not OrderedDict). This function takes one parameter, the name of the file. Your function should convert the number values for wins and nomination from string to integer as you are reading the content from the file. When the function is called, it will produce the following result.

```
grammyWinners = getGrammyWinners('grammy-winners.csv')
print(grammyWinners[:3])

[{'nominee': 'Quincy Jones', 'sex': 'M', 'wins': 28, 'nominations': 80},
{'nominee': 'Beyoncé', 'sex': 'F', 'wins': 28, 'nominations': 79},
{'nominee': 'Alison Krauss', 'sex': 'F', 'wins': 27, 'nominations': 42}]
```

Part 2: Print female artists

Write the function `printFemaleArtists`, which takes one parameter, the list of dictionaries from Part 1, and prints out the following:

There are 12 female artists among the top 50 Grammy winners.
These female artists are:
Beyoncé with 28 wins and 79 nominations.
Alison Krauss with 27 wins and 42 nominations.
Aretha Franklin with 18 wins and 44 nominations.

Alicia Keys with 15 wins and 29 nominations.
Adele with 15 wins and 18 nominations.
Leontyne Price with 13 wins and 25 nominations.
Ella Fitzgerald with 13 wins and 20 nominations.
Lady Gaga with 12 wins and 29 nominations.
CeCe Winans with 12 wins and 28 nominations.
Dixie Chicks with 12 wins and 19 nominations.
Taylor Swift with 11 wins and 41 nominations.
Shirley Caesar with 11 wins and 28 nominations.

Part 3: Sort by success rate
The provided CSV file ranks artists based on the total number of wins. Another way to rank them would be by success rate (the percentage of wins out of all nominations). You will write a function, `sortBySuccessRate`, that does exactly that. It takes as a parameter the list of dictionaries from Part 1 and returns the list of sorted dictionaries by success rate. Hint: create a new key:value pair to store the success rate value. Use a helper function to be used in the **sorted** function.

```
>> sortBySuccessRate(grammyWinners)[:3]
[{'nominee': 'Adele',
  'sex': 'F',
  'wins': 15,
  'nominations': 18,
  'successRate': 0.83},
 {'nominee': 'Jimmy Sturr',
  'sex': 'M',
  'wins': 18,
  'nominations': 24,
  'successRate': 0.75},
 {'nominee': 'T Bone Burnett',
  'sex': 'M',
  'wins': 13,
  'nominations': 18,
  'successRate': 0.72}]
```

Bonus Task: Save the result from calling the function `sortBySuccessRate` into a JSON file, titled, `successfulArtists.json`. What do you see as the benefit of storing the list of dictionaries into a JSON file as compared to a CSV file?

Problem 5: countQs

[involves dictionaries, loops]

Define a function called `countQs` that takes a dictionary as an argument. The keys of the dictionary are strings, and the values are lists of strings. Your function should return the total number of keys and list entries that contain question marks. See examples below.

Invocation	Return Value
`countQs({'me?':['you!','you?']})`	2
`countQs({'person':['who?','whom?'], 'place':['where?']})`	3
`countQs({'why me???':['because','ok?']})`	2
`countQs({'nope':[]})`	0

Define your `countQs` function in the box below:

Problem 6: Comprehending Comprehensions

Rewrite the function below using a list comprehension:

```
def capitalBookLengths(books):
    lengths = []
    for b in books:
        if b[0] == b[0].upper():
            lengths.append(len(b))
    return lengths
```

Problem 7: Mass Municipalities

[involves dictionaries, files, list comprehensions, sorting]

This problem involves analysis of municipality data for Massachusetts taken from https://en.wikipedia.org/wiki/List_of_municipalities_in_Massachusetts.

Assume you are given a dictionary `massMunics` that summarizes some of this data. Here is an example of some of the key-value pairs in this dictionary:

```
massMunics = {
  "Boston": {"county": "Suffolk", "population": 617660, "type": "City", "year": 1630},
  "Cambridge": {"county": "Middlesex", "population": 105162, "type": "City", "year": 1636},
  "Gosnold": {"county": "Dukes", "population": 75, "type": "City", "year": 1864},
  "Natick": {"county": "Middlesex", "population": 33006, "type": "Town", "year": 1781},
  "Needham": {"county": "Norfolk", "population": 28886, "type": "Town", "year": 1711},
  "Newton": {"county": "Middlesex", "population": 85146, "type": "City", "year": 1688},
  # In names normally with spaces, like "New Bedford", use underscore instead of space.
  "New_Bedford": {"county": "Bristol", "population": 95072, "type": "City", "year": 1787},
  "Wellesley": {"county": "Norfolk", "population": 27982, "type": "Town", "year": 1881},
  "Wellfleet": {"county": "Barnstable", "population": 2750, "type": "Town", "year": 1775},
  …
}
```

Part 7a

Download the files munics.py and massMunics.py into the same folder. Then flesh out the following function definitions according to their docstrings. Notes:

- List comprehensions are your friends here!
- For sorting, define helper functions that you can use as the `key` parameter.

```
def listByNameLength(munics, n):
    '''Return a list of the top n municipalities, in descending order by
    name length. Municipalities with the same name length should be in
    reverse alphabetical order.
    E.g. listByNameLength(massMunics, 10) returns
    ['Manchester-by-the-Sea', 'North_Attleborough', 'West_Stockbridge',
     'West_Springfield', 'West_Bridgewater', 'North_Brookfield',
     'Mount_Washington', 'Great_Barrington', 'East_Bridgewater', 'West_Brookfield']
   '''

def listByYear(munics, n):
    '''Return a list of the first n pairs (municipality, year) in ascending order
    by year. Municipalities with the same year should be in alphabetical order.
    E.g. listByNameYear(massMunics, 10) returns
    [('Plymouth', 1620), ('Gloucester', 1623), ('Salem', 1626), ('Lynn', 1629),
     ('Marblehead', 1629), ('Boston', 1630), ('Medford', 1630),
     ('Watertown', 1630), ('Ipswich', 1634), ('Concord', 1635)]'''
```

```
def largestTown(munics):
    '''Return the pair (town, population) for the municipality of type 'Town'
    that has the largest population. E.g. largestTown(massMunics) returns
    ('Brookline', 58732)'''

def smallestCity(munics):
    '''Return the pair (city, population) for the municipality of type 'City'
       that has the smallest population. E.g. largestTown(massMunics) returns
       ('Palmer', 12140)'''

def writeByPopulation(filename, munics, n):
    '''In the given filename, write the top n municipalities in descending order
    by population. Each line should have the form:
       name: population
    E.g., writeByPopulation('massMunicsByPop.txt', massMunics, 5) creates a file
    named massMunicsByPop.txt with 6 lines of content:
    Boston: 617660
    Worcester: 181045
    Springfield: 153060
    Lowell: 106519
    Cambridge: 105162
    New_Bedford: 95072'''
def listCitiesSmallerThanLargestTown(munics):
    '''Return a list of (city, population) pairs, in ascending order by population,
       of all cities whose population is less than the population of the largest
       Town. E.g. largestTown(massMunics) returns
       [('Palmer', 12140), ('North_Adams', 13708), ('East_Longmeadow', 15720),
        ('Easthampton', 16053), ('Amesbury', 16283), ('Southbridge', 16719),
        ('Newburyport', 17416), ('Greenfield', 17456), ('Winthrop', 17497),
        ('Gardner', 20228), ('Bridgewater', 26563), ('Melrose', 26983),
        ('West_Springfield', 28391), ('Agawam', 28438), ('Northampton', 28549),
        ('Gloucester', 28789), ('Franklin', 31635), ('Watertown', 31915),
        ('Randolph', 32112), ('Chelsea', 35177), ('Braintree', 35744),
        ('Woburn', 38120), ('Marlborough', 38499), ('Beverly', 39502),
        ('Holyoke', 39880), ('Fitchburg', 40318), ('Leominster', 40759),
        ('Westfield', 41094), ('Salem', 41340), ('Everett', 41667),
        ('Attleboro', 43593), ('Pittsfield', 44737), ('Barnstable', 45193),
        ('Methuen', 47255), ('Peabody', 51251), ('Revere', 51755),
        ('Weymouth', 53743), ('Chicopee', 55298), ('Taunton', 55874),
        ('Medford', 56173)]'''


def listCountiesByPopulation(munics):
    '''Return a list of (county, population) pairs, in descending order by
    population, of all counties, where the population of a county is determined by
    the sum of the municipalities in that county.
    E.g., listCountiesByPopulation(massMunics) returns
```

```
[('Middlesex', 1481036), ('Worcester', 798552), ('Essex', 743159),
 ('Suffolk', 722089), ('Norfolk', 670850), ('Bristol', 548285),
 ('Plymouth', 494919), ('Hampden', 463490), ('Barnstable', 215888),
 ('Hampshire', 158080), ('Berkshire', 131219), ('Franklin', 71372),
 ('Dukes', 16535), ('Nantucket', 10172)]'''
```

Part 7b

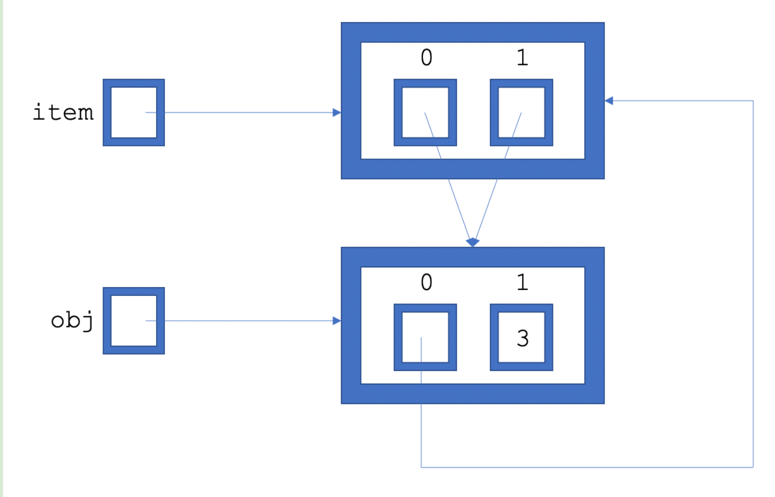
The file massMunic.txt contains the Massachusetts municipality data, with one line per municipality, in the following format:

```
Abington              Town    Plymouth     15985   1712
Acton                 Town    Middlesex    21924   1735
Acushnet              Town    Bristol      10303   1860
Adams                 Town    Berkshire    8485    1778
Agawam                City    Hampden      28438   1855
Alford                Town    Berkshire    494     1773
Amesbury              City    Essex        16283   1668
Amherst               Town    Hampshire    37819   1775
```

Define a function `readMunicFile` that takes the name of a file whose format is like massMunic.txt and returns a dictionary of the form `massMunics` used in Part 2a. Note that the population and year components are integers, not strings.

Problem 8 : Memory Diagrams

Write a snippet of Python code that produces the memory diagram below.



Assume that you start with the following code:

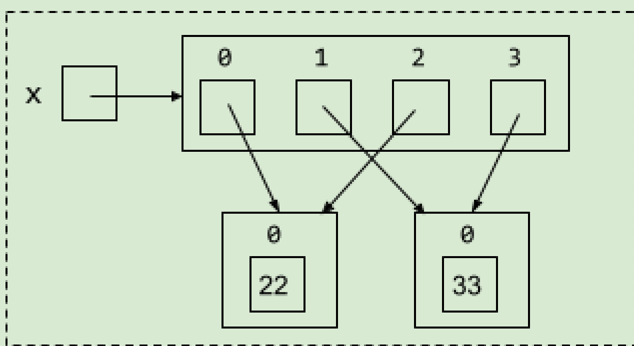
```
obj = [[3], 7]
item = [obj]
```

When writing your code, you may not declare any new lists. You must only modify the two lists given. Feel free to use the methods `.pop`, `.append`, and `.insert` to mutate either list.

Problem 9 : Memory Diagrams

Part a: Matching Code Snippets to Memory Diagrams

Below are 20 code snippets that attempt to create the following memory diagram:



Almost all of these snippets were submitted by students as solutions to this problem on a midterm exam during a previous semester. (In some cases, they have been edited in minor ways.)

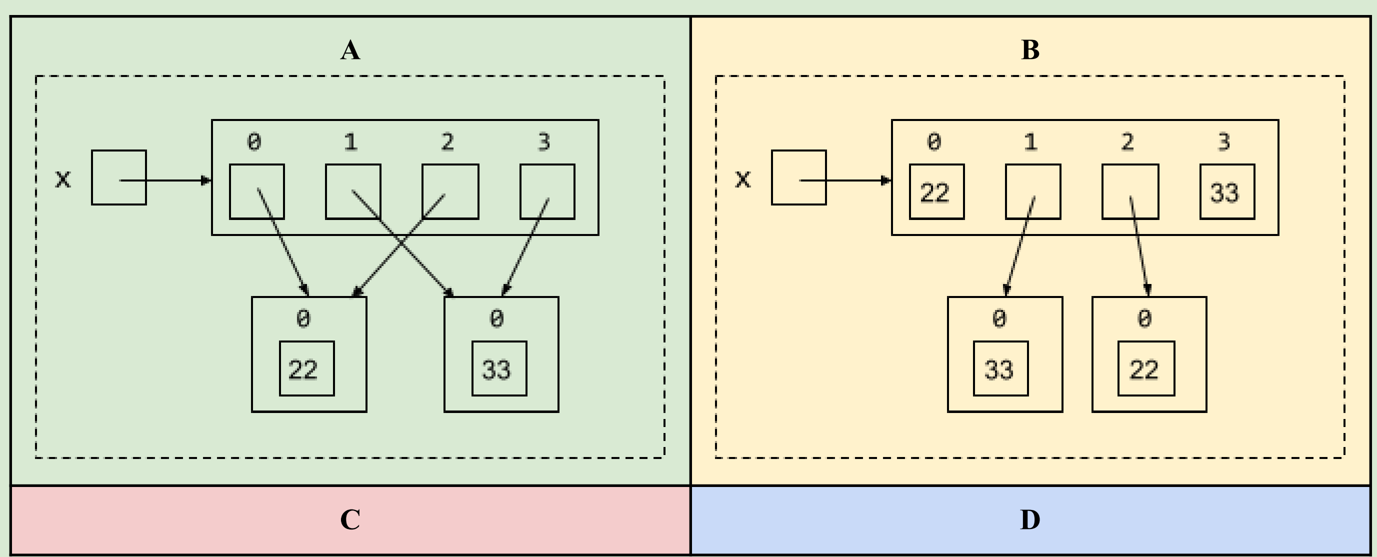
Your goal is to match each of the 20 code snippets with corresponding memory diagrams that are shown below the snippets. (For convenience, you the memory diagrams are also available on on this other page so you can view them side-by-side with the snippets .) Each of the code snippets should generate one of the 10 diagrams. Which one? The same diagram may be generated by multiple snippets.

Code Snippets

CODE01 a = [22] b = [33] x = [a, b, a, b]	CODE02 x = [[22], [33], [22], [33]] x[0][0] = x[2] x[3][0] = x[1]	CODE03 x = [[22], [33], [22], [33]]
CODE04 x = [[22], 5, 6, [33]] x[2] = x[0] x[1] = x[3]	CODE05 x = [[22], [33], [22], [33]] x[2][0] = x[0] x[3][0] = x[1]	CODE06 x = [[22], [33], [22], [33]] x[2] = x[0][0] x[3] = x[1][0]

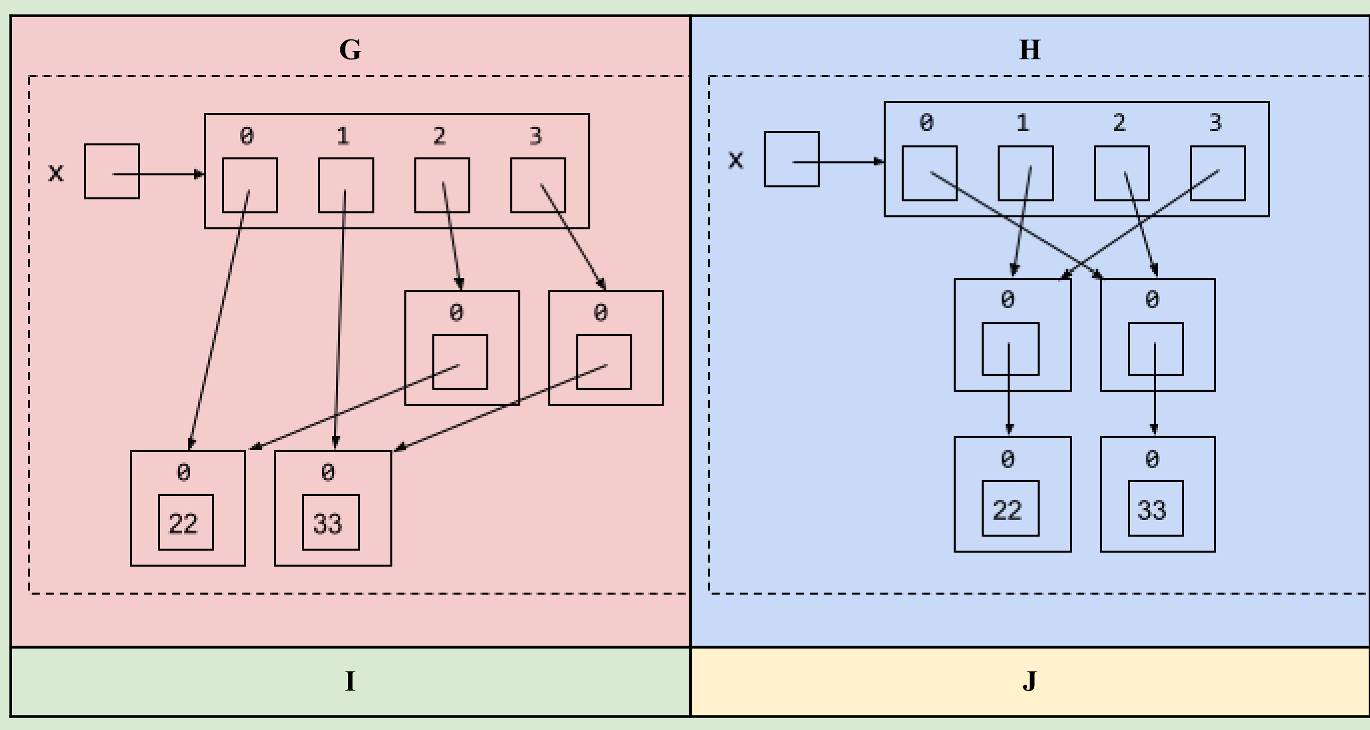
CODE07 x = [[22], [33], [22], [33]] x[0] = x[2] x[1] = x[3]	CODE08 x = [[22], [33], [22], [33]] x[0][0] = x[2][0] x[1][0] = x[3][0]	CODE09 x = [[22], [33], 5, 6] x[2] = x[0] x[3] = x[1]
CODE10 x = [[22], 5, 6, [33]] x[1] = x[3][0] x[2] = x[0][0]	CODE11 x = [[22], [33]] x = x*2	CODE12 r = [[22]] s = [[33]] x = [s, r, s, r]
CODE13 x = [22, 33] x.append(x[0]) x.append(x[1])	CODE14 x = [[22], [33]] x.append(x[0]) x.append(x[1])	CODE15 x = [[22], [33]] x.append(x)
CODE16 x = [[22], [33]] x.append(0) x.append(1) x[2] = x[0] x[3] = x[1]	CODE17 x = [[33], [22]] x.insert(0, x[1][0]) x.append(x[1][0])	CODE18 x = [[22], [33]] x.insert(1, x[1][0]) x.insert(2, x[0][0])
CODE19 x = [[22], [33], [22], [33]] x[2] = x[0] x[3] = x[1]	CODE20 x = [[[22]], [[33]], [[22]], [[33]]] x[1] = x[3][0] x[2] = x[0][0]	

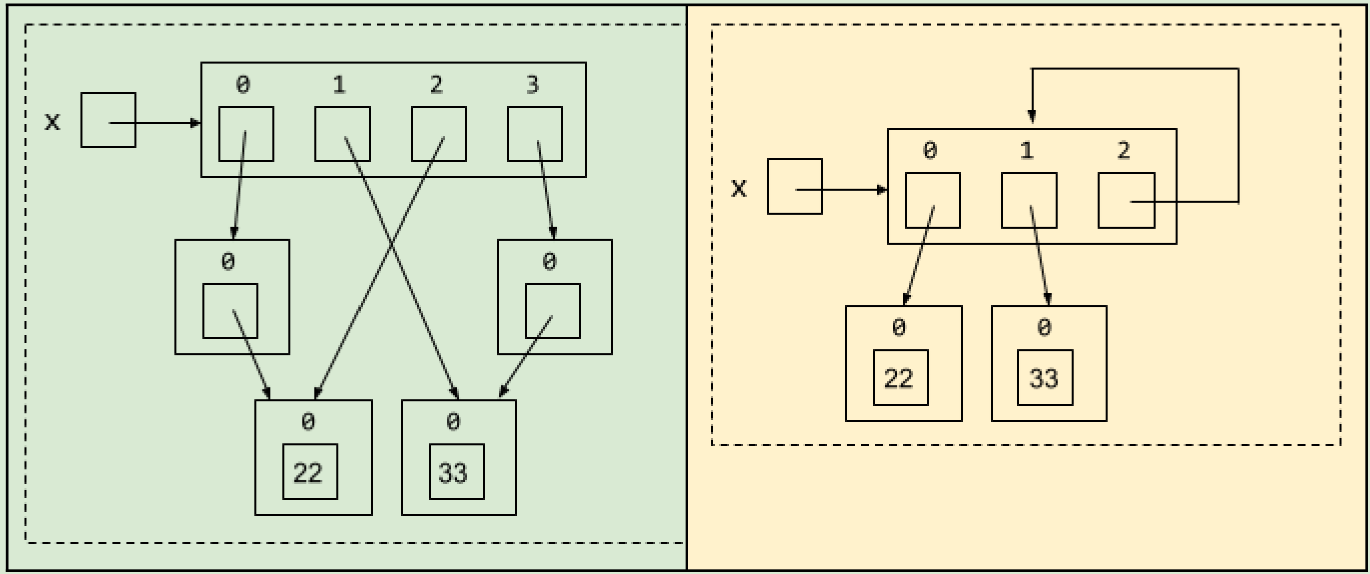
Memory Diagrams



E

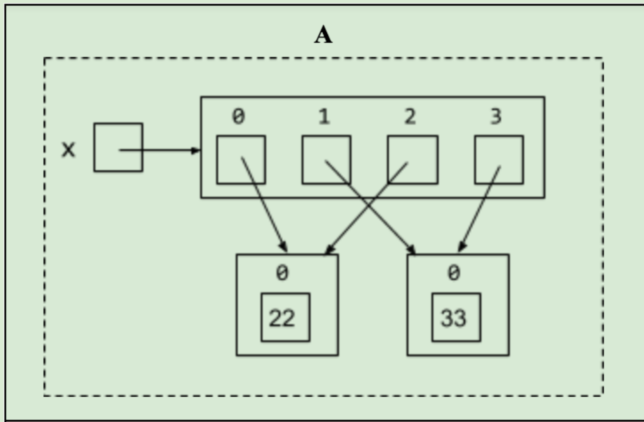
F



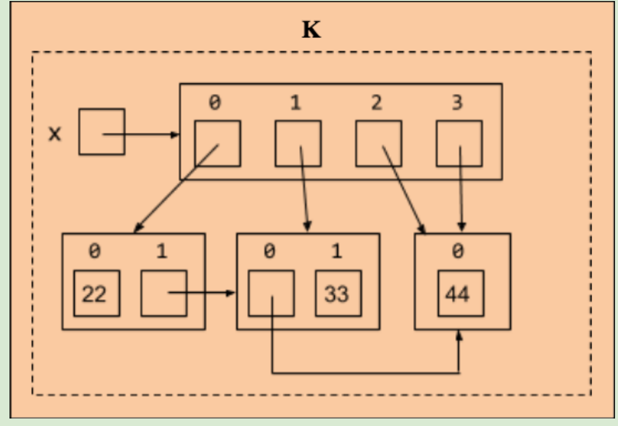


Part b: Modifying Memory Diagram A

Assume that the value of variable `x` is the list with memory diagram **A**:



In this part, you will flesh out the body of the function `A_to_K` that takes as its single parameter a list named `L` and modifies it in such a way that after calling `A_to_K(x)`, the structure of `x` has been changed to that of memory diagram **K**

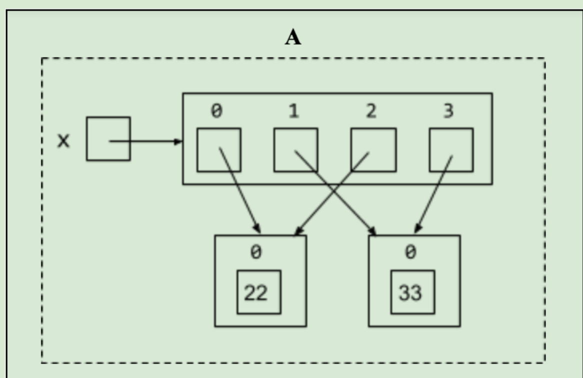


In the body of the `A_to_K` function, your code should satisfy the following restrictions:

- The **only** local variable it is allowed to use is the parameter `L`. It must **not** refer to the variable `x`, nor should it introduce any other local variable names. (But because you may assume that the `A_to_K` function is called on `x`, `x` and `L` will point to the very same list.)
- It must create exactly **one** new list, the singleton list whose only list slot contains 44.
- It must **not** change the contents of any existing list slot containing a number. So the list with a slot containing 22 in diagram **K** must be a modified version of the list with the slot containing 22 in diagram **A**, and similarly for the list containing 33.
- It **must** change the contents of some existing list slots that do not contain numbers.
- Your code may create new list slots in existing lists by calling the `.append` and `.insert` methods.

Part c: Drawing a Modified Memory Diagram

Assume that the value of variable `x` is the list with memory diagram **A**:



In this part, draw the **final** memory diagram that shows the structure of the list `x` after executing the following sequence of statements:

```
x[3] = [x[1], x[0], x[2][0]]
x[1].insert(0, x[2])
x[0].append(56)
x[2][0] = x[2][1] + x[1][1] + x[3][0][0][0]
```

Solutions to Review Problems

Solution to Problem 0: Interpreting Functions

```
meow
meow
chirpbark
```

Solution to Problem 1: Nested loops

```
def countDirectFlights(data):
    count = 0
    for flightList in data:
        for flight in flightList:
            if flight == "D":
                count +=1
    return count
```

Solution to Problem 2: Reading and Writing From Files

Part A:

```
## Three solutions to readEvents are presented below
def readEvents1(filename):
    with open(filename, 'r') as inputFile:
        events = []
        for line in inputFile:
            event = line.strip().split('-')
            # strip() removes trailing newline (so that name doesn't include
newline)
            # event is list of four elements
            events.append(tuple(event)) # tuple converts list to tuple
        return events

def readEvents2(filename):
    with open(filename, 'r') as inputFile
        events = []
        for line in inputFile:
            # use tuple assignment to name parts of list
            year, month, day, name = line.strip().split('-')
            # strip() removes trailing newline (so that name doesn't include
newline)
            # Double parens needed: outer for .append call; inner for tuple.
            events.append( (year, month, day, name) ) # tuple converts list to tuple
        return events

# This version uses a list comprehension
def readEvents3(filename):
    with open(filename, 'r') as inputFile:
        return [tuple(line.strip().split('-')) for line in inputFile]
```

Part B:

```
## Five solutions to writeEvents are presented below
def writeEvents1(filename, events):
```

```
    with open(filename, 'w') as outputFile:
        for evt in events: # evt is a tuple of 4 strings
            outputFile.write("-".join(evt) + "\n")

def writeEvents2(filename, events):
    with open(filename, 'w') as outputFile:
        # use tuple assignment to name parts of tuple
        for year, month, day, name in events:
            outputFile.write(f"{year}-{month}-{day}-{name}\n")

def writeEvents3(filename, events):
    with open(filename, 'w') as outputFile:
        # use tuple assignment to name parts of tuple
        for year, month, day, name in events:
            outputFile.write(year + "-" + month + "-"
                             + day + "-" + name + "\n")

def writeEvents4(filename, events):
    with open(filename, 'w') as outputFile:
        for evt in events:
            outputFile.write(f"{evt[0]}-{evt[1]}-{evt[2]}-{evt[3]}\n")

def writeEvents5(filename, events):
    with open(filename, 'w') as outputFile:
        for evt in events:
            outputFile.write(evt[0] + "-" + evt[1] + "-"
                             + evt[2] + "-" + evt[3] + "\n")
```

Solution to Problem 3: Working with dictionaries and lists

Part a - Alternative 1: (uses list comprehension)

```
def getYear(data):
    return data['year']

def booksSortedByDate(authors, name):
    # Notice the use of list comprehension
    booksForAuthor = [entry for entry in authors if entry['author'] == name]
    result = ""
    sortedBooks = sorted(booksForAuthor, key = getYear)
    for book in sortedBooks:
        result += f"{book['author']}: {book['book']}, {book['year']}\n"
    return result
```

Part b

```
def groupByAuthors(authorsList):
    """Solution for Part b"""
    authorsDict = {}
    for entry in authorsList:
        if entry['author'] in authorsDict:
            authorsDict[entry.pop('author')].append(entry)
        else:
            authorsDict[entry.pop('author')] = [entry]
    return authorsDict
```

Notice that this function is mutating the dictionaries stored in `authorsList`, thus, if you invoke this with the object authors:

```
groupByAuthors(authors)
```

it will have the side effect of changing the list authors, to not contain the field 'author' anymore. Concretely, now the list will look like this:

```
[{'book': 'Persuasion', 'year': 1818},
 {'book': 'The Voyage Out', 'year': 1915},
 {'book': 'Emma', 'year': 1815},
 {'book': 'Mrs Dalloway', 'year': 1925},
 {'book': 'Orlando', 'year': 1928},
 {'book': 'Pride and Prejudice', 'year': 1813},
 {'book': 'Night and Day', 'year': 1919},
 {'book': 'Sense and Sensibility', 'year': 1811},
 {'book': 'To the Lighthouse', 'year': 1927},
 {'book': 'Northanger Abbey', 'year': 1818},
 {'book': 'The Waves', 'year': 1931},
 {'book': 'Mansfield Park', 'year': 1814}]
```

To avoid this side-effect, we will need to make something called a deepcopy of our list, which doesn't refer to the same dict objects, but makes copies of each of them.

```
import copy
groupByAuthors(copy.deepcopy(authors))
```

Solution to Problem 4: Working with CSV files and dictionaries

Part 1: Get the Grammy winners

Notice that the following function reads each dictionary one by one from the opened file and converts the values to integers before appending the dictionary to the list of results.

```
def getGrammyWinners(filename):
    """Read the content of a CSV file and return a list of dictionaries.
```

```
    """
    allArtists = []
    with open(filename,'r') as inputF:
        dictReader = csv.DictReader(inputF)
    for artistDct in dictReader:
         artistDct['wins'] = int(artistDct['wins'])
         artistDct['nominations'] = int(artistDct['nominations'])
         allArtists.append(dict(artistDct)) # convert to classic
dictionary
    return allArtists

grammyWinners = getGrammyWinners('grammy-winners.csv')
print(grammyWinners[:3])
```

Part 2: Print female artists

```
def printFemaleArtists(artists):
    """Find the female artists, then print out the results.
    """
    femaleArtists = [artistDct for artistDct in artists if
                     artistDct['sex']=='F']
    print(f"There are {len(femaleArtists)} female artists " +\
          f"among the top {len(artists)} Grammy winners.")
    print("These female artists are:")
    for fA in femaleArtists:
    print(f"{fA['nominee']} with {fA['wins']} wins and {fA['nominations']}
nominations.")
```

Part 3: Sort artists by success rate

```
def bySuccessRate(artistDct):
    """Helper function for sorting"""
    return artistDct['successRate']

def sortBySuccessRate(artists):
    """First create a new key:value pair to record the success rate,
    then sort the list of artists by the success rate.
    """
    for artistDct in artists:
        artistDct['successRate']= round(artistDct['wins'] /
                                        artistDct['nominations'], 2)

    sortedArtists = sorted(artists, key=bySuccessRate, reverse=True)
    return sortedArtists
```

Solution for Problem 5: countQs

```
def countQs(d):
      total = 0
      for k in d:
            if "?" in k:
                  total += 1
            for item in d[k]:
                  if "?" in item:
                        total +=1
      return total
```

Solution for Problem 6: Comprehending Comprehensions

```
def capitalBookLengths(books):
    return [len(b) for b in books if b[0] == b[0].upper()]
```

Solution for Problem 7: Mass Municipalities

Part 7a Solutions

```
def listByNameLength(munics, n):
    '''Return a list of the top n municipalities, in descending order by
       name length. Municipalities with the same name length should be in
       reverse alphabetical order.'''
    return sorted(munics,
                  key=lenThenAlphabetical,
                  reverse=True)[:n]

def lenThenAlphabetical(name):
      '''Return a tuple of (1) the length of the name and (2) the name.
         This can be used to sort data first by name length,
         using alphabetical order as a tiebreaker.'''
      return (len(name), name)

def listByYear(munics, n):
    '''Return a list of the first n pairs (municipality, year) in ascending order
       by year. Municipalities with the same year should be in alphabetical
order.'''
    return sorted([(mun, munics[mun]['year']) for mun in munics],
                  key=yearThenName,
                  )[:n]
```

```
def yearThenName(nameYearPair):
    '''Given a pair (2-tuple) of municipality name and year of founding,
         return a pair of the year and name. Used for sorting where
         year matters most.'''
      name, year = nameYearPair
      return (year, name)

def largestTown(munics):
    '''Return the pair (town, population) for the municipality of type 'Town'
       that has the largest population.'''
    pop,town = max([(munics[m]['population'], m) for m in munics
                    if munics[m]['type'] == 'Town'])
    return (town, pop)

def smallestCity(munics):
    '''Return the pair (city, population) for the municipality of type 'City'
       that has the smallest population.'''
    pop,town = min([(munics[m]['population'], m) for m in munics
                    if munics[m]['type'] == 'City'])
    return (town, pop)

def writeByPopulation(filename, munics, n):
    '''In the given filename, write the top n municipalities in descending order
       by population. Each line should have the form:
       name: population'''
    with open(filename, 'w') as outfile:
        for (mun, pop) in sorted([(mun, munics[mun]['population'])
                                  for mun in munics],
                                  key=getPopulation,
                                  reverse=True)[:n]:
            outfile.write(f'{mun}: {pop}\n')

def getPopulation(entityPopulationPair):
    '''Given a pair (2-tuple) of an entity and its population.
       return just the population. Used for sorting by population.'''
    return entityPopulationPair[1]

def listCitiesSmallerThanLargestTown(munics):
    '''Return a list of (city, population) pairs, in ascending order by population,
       of all cities whose population is less than the population of the largest
town.'''
    largestTownPop = max([munics[m]['population'] for m in munics
                          if munics[m]['type'] == 'Town'])
    return sorted([(m, munics[m]['population']) for m in munics
                   if munics[m]['type'] == 'City'
                      and munics[m]['population'] < largestTownPop],
```

```
                  key=getPopulation)

def listCountiesByPopulation(munics):
    '''Return a list of (county, population) pairs, in descending order by
population,
       of all counties, where the population of a county is determined by the sum of
       the municipalities in that county.'''
    countiesWithDups = [munics[m]['county'] for m in munics]
    countiesWithoutDups = []
    for county in countiesWithDups:
        if county not in countiesWithoutDups:
            countiesWithoutDups.append(county)
    countyPops = [(c, sum([munics[m]['population']
                           for m in munics if munics[m]['county'] == c]))
                   for c in countiesWithoutDups]
    return sorted(countyPops, key=getPopulation, reverse=True)

# Alternative solution for listCountiesByPopulation
def listCountiesByPopulation_v2(munics):
    '''Return a list of (county, population) pairs, in descending order by
population,
       of all counties, where the population of a county is determined by the sum of
       the municipalities in that county.'''
    countyPopDict = {}
    for m in munics:
        county = munics[m]['county']
        municPop = munics[m]['population']
        countyPopDict[county] = municPop + countyPopDict.get(county, 0)
    return sorted(countyPopDict.items(), key=getPopulation, reverse=True)
```

Part 7b Solution

```
def readMunicFile(filename):
    with open(filename, 'r') as infile:
        dct = {}
        for line in infile:
            muni, type, county, pop, year = line.split()
            dct[muni] = {'type': type, 'county': county,
                         'population': int(pop), 'year': int(year)}
        return dct
```

Solution for Problem 8 : Memory Diagrams

obj.pop(1)
obj[0].append(10)
item.append(obj[0])

Solution for Problem 9 : Memory Diagrams

Part 9a: Matching Code Snippets to Memory Diagrams

Below we show which of the 20 given code snippets make each diagram.

CODE01
```
a = [22]
b = [33]
x = [a, b, a, b]
```

CODE04
```
x = [[22], 5, 6, [33]]
x[2] = x[0]
x[1] = x[3]
```

CODE07
```
x = [[22], [33], [22], [33]]
x[0] = x[2]
x[1] = x[3]
```

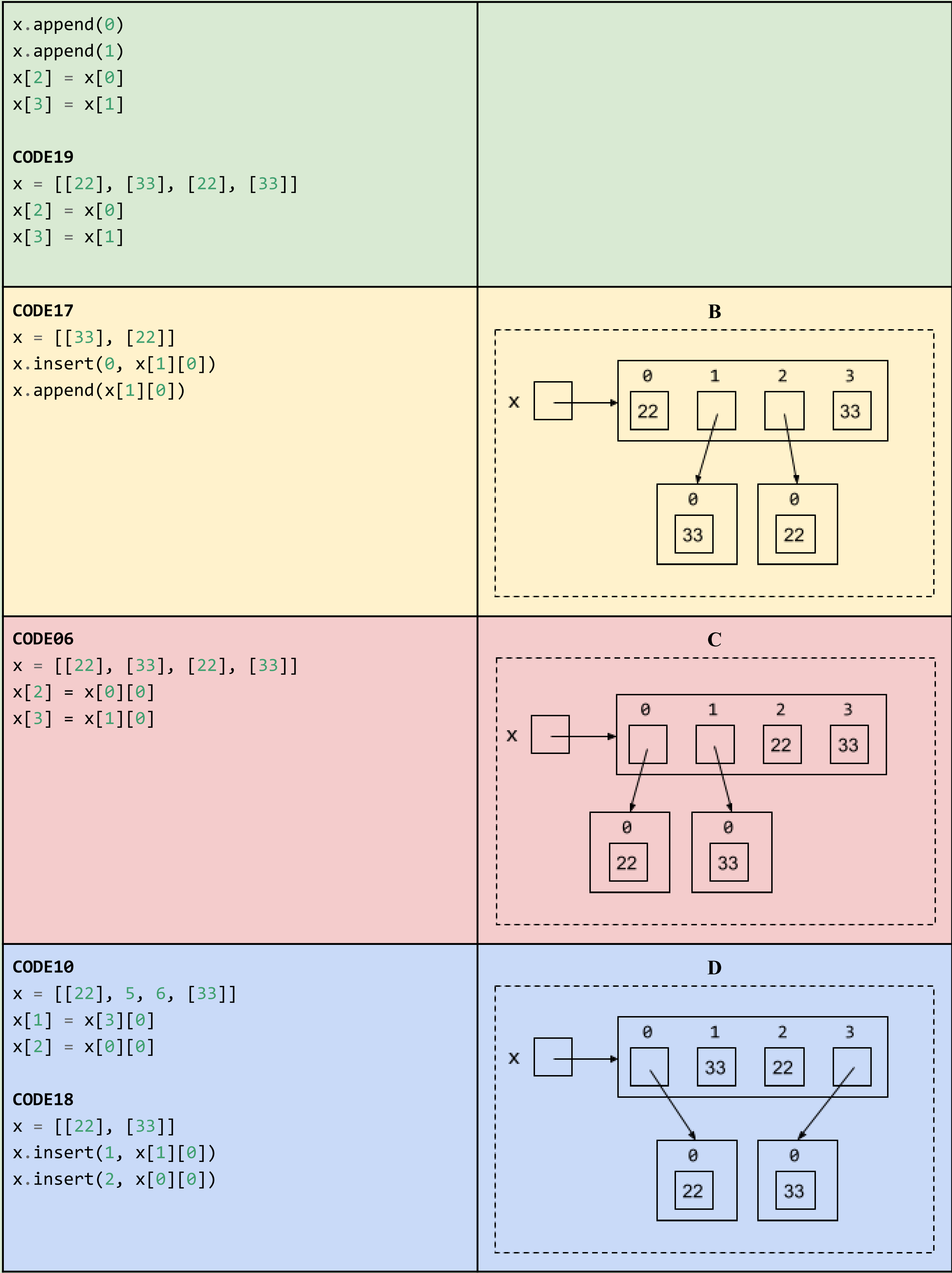
CODE09
```
x = [[22], [33], 5, 6]
x[2] = x[0]
x[3] = x[1]
```

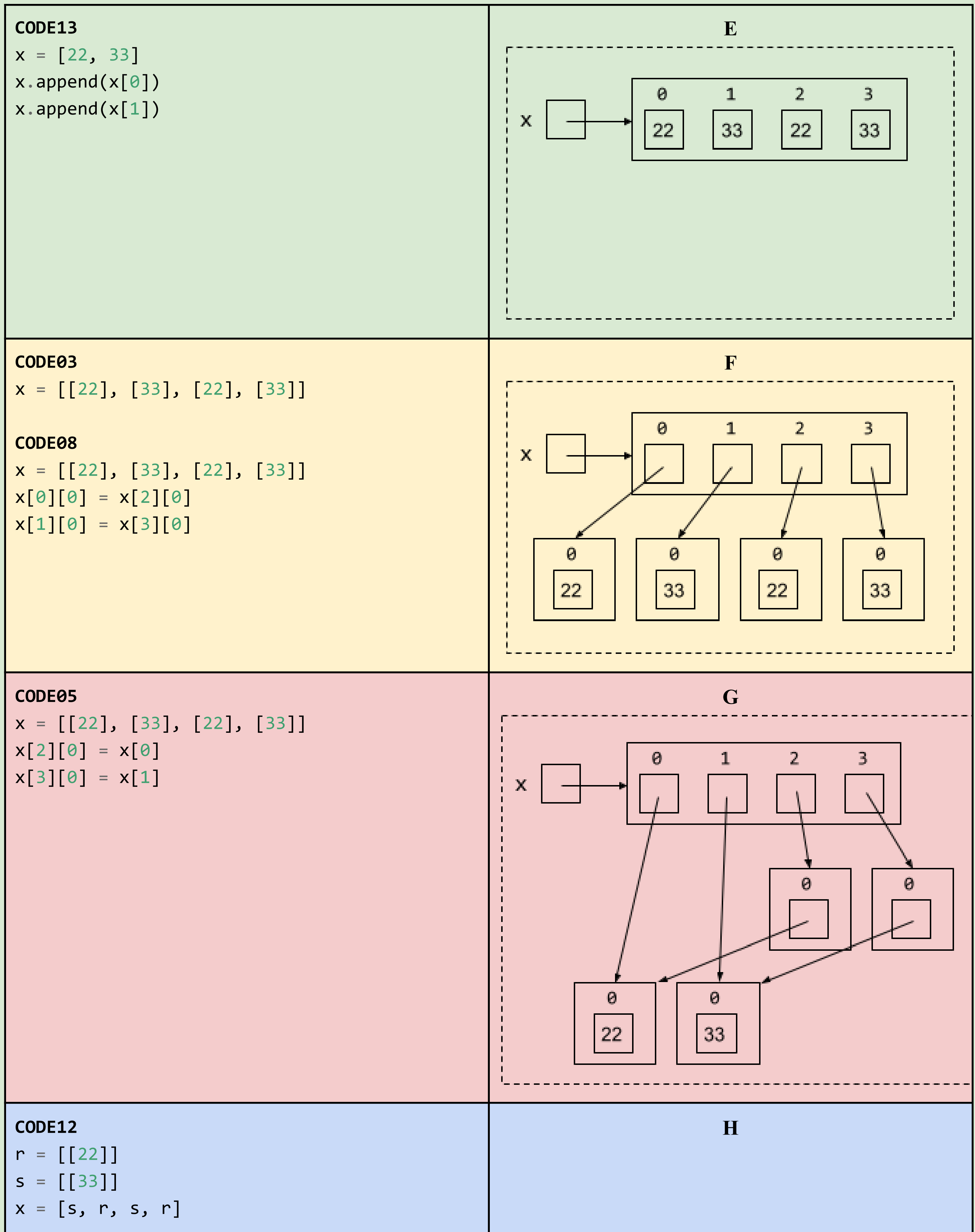
CODE11
```
x = [[22], [33]]
x = x*2
```

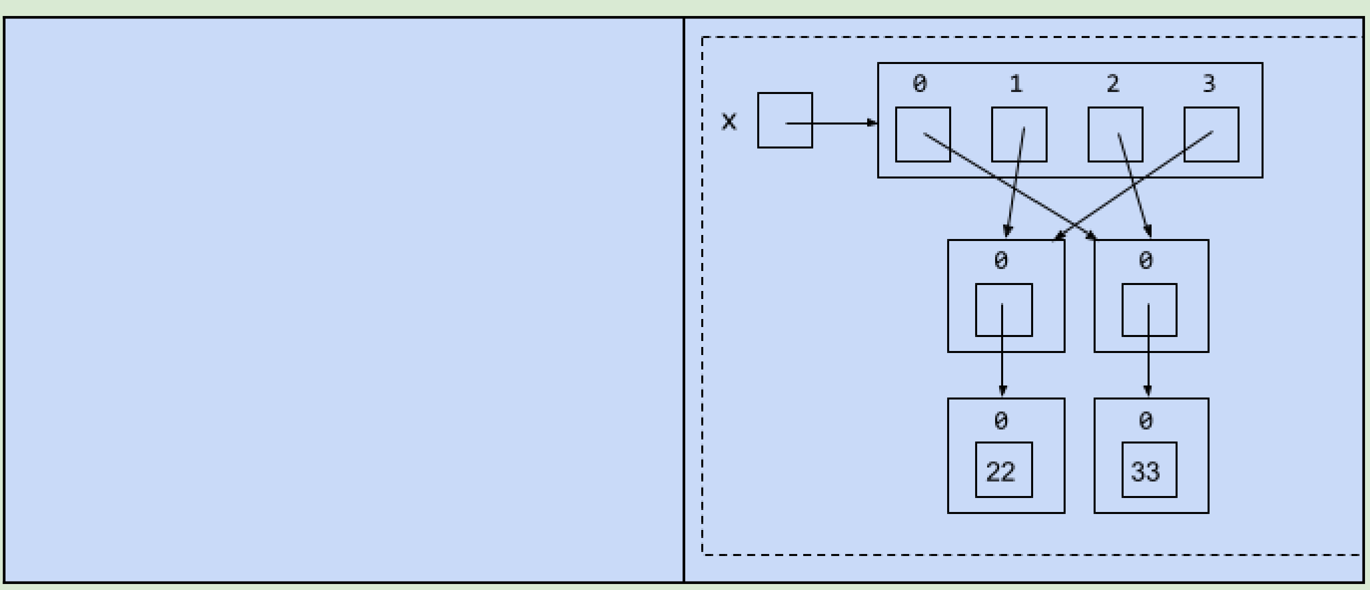
CODE14
```
x = [[22], [33]]
x.append(x[0])
x.append(x[1])
```

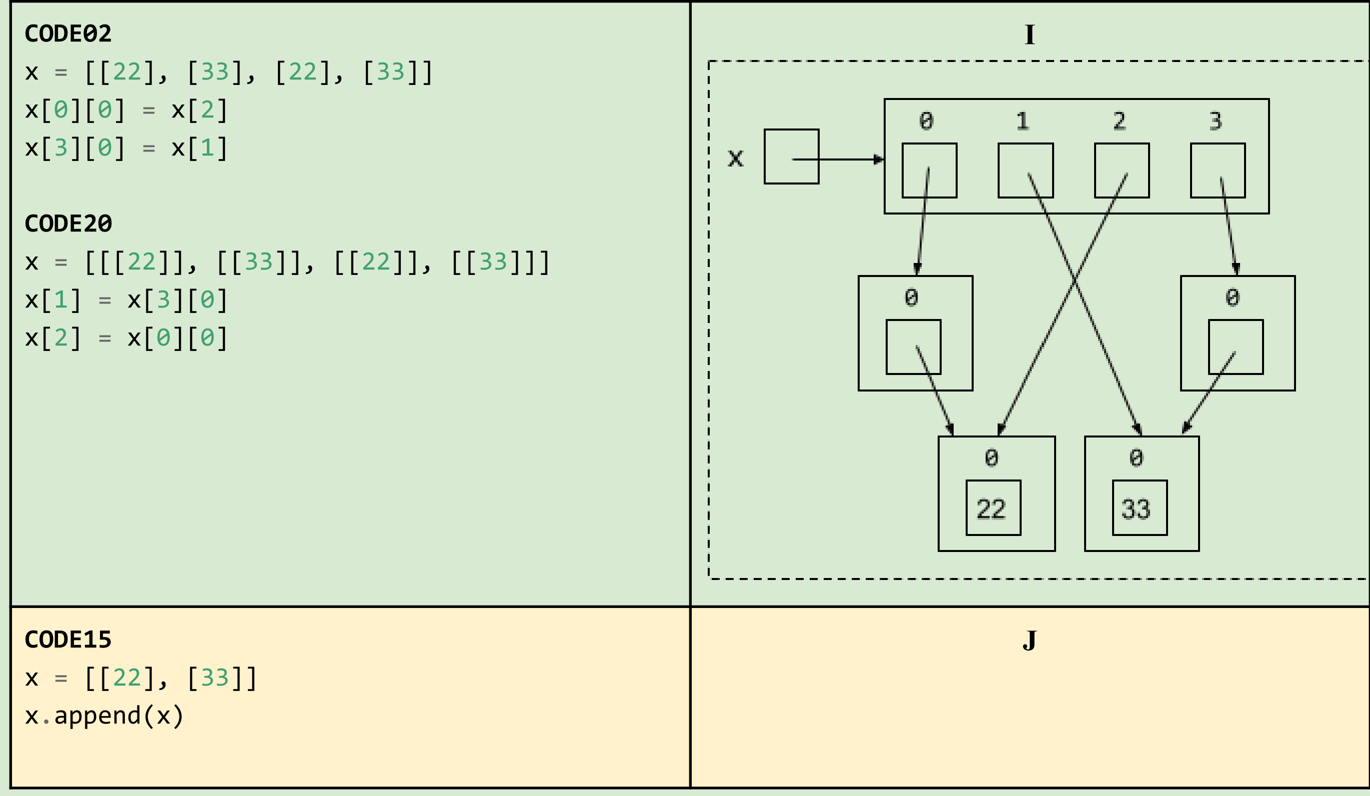
CODE16
```
x = [[22], [33]]
```

A


x.append(0)
x.append(1)
x[2] = x[0]
x[3] = x[1]
CODE19
x = [[22], [33], [22], [33]]
x[2] = x[0]
x[3] = x[1]
CODE17
x = [[33], [22]]
x.insert(0, x[1][0])
x.append(x[1][0])
B
x
0 1 2 3
22
33
0
33
0
22
CODE06
x = [[22], [33], [22], [33]]
x[2] = x[0][0]
x[3] = x[1][0]
C
x
0 1 2 3
22
33
0
22
0
33
CODE10
x = [[22], 5, 6, [33]]
x[1] = x[3][0]
x[2] = x[0][0]
CODE18
x = [[22], [33]]
x.insert(1, x[1][0])
x.insert(2, x[0][0])
D
x
0 1 2 3
33
22
0
22
0
33


CODE13
x = [22, 33]
x.append(x[0])
x.append(x[1])
E
x
0 1 2 3
22 33 22 33
CODE03
x = [[22], [33], [22], [33]]
CODE08
x = [[22], [33], [22], [33]]
x[0][0] = x[2][0]
x[1][0] = x[3][0]
F
x
0 1 2 3
0 22
0 33
0 22
0 33
CODE05
x = [[22], [33], [22], [33]]
x[2][0] = x[0]
x[3][0] = x[1]
G
x
0 1 2 3
0
0
0 22
0 33
CODE12
r = [[22]]
s = [[33]]
x = [s, r, s, r]
H





CODE02

```
x = [[22], [33], [22], [33]]
x[0][0] = x[2]
x[3][0] = x[1]
```

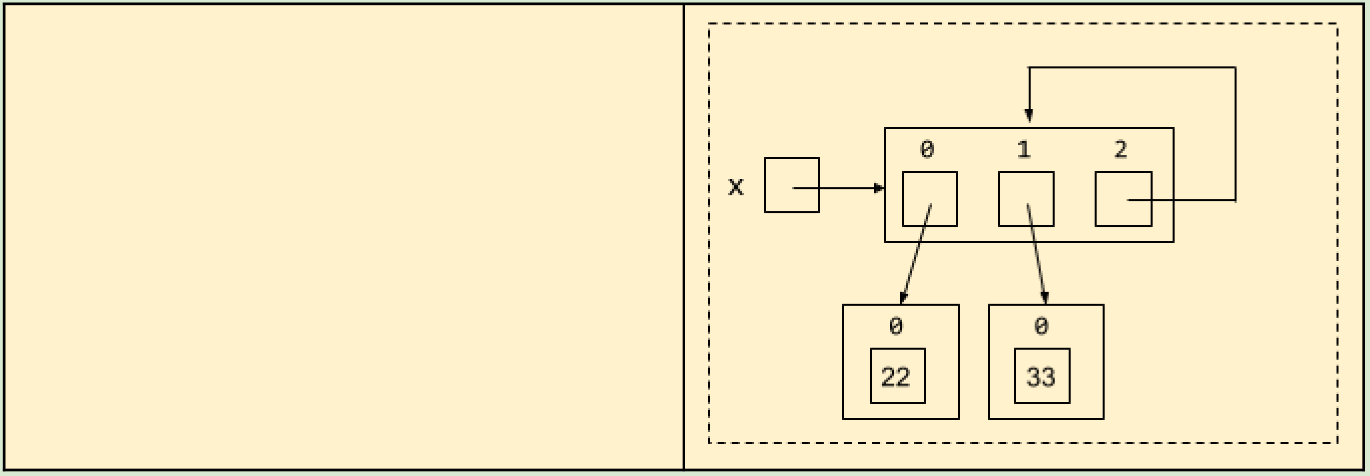
CODE20

```
x = [[[22]], [[33]], [[22]], [[33]]]
x[1] = x[3][0]
x[2] = x[0][0]
```


CODE15

```
x = [[22], [33]]
x.append(x)
```

J



Based on the above associations between snippets and diagrams, here are the correct assignments of code snippet to memory diagram:

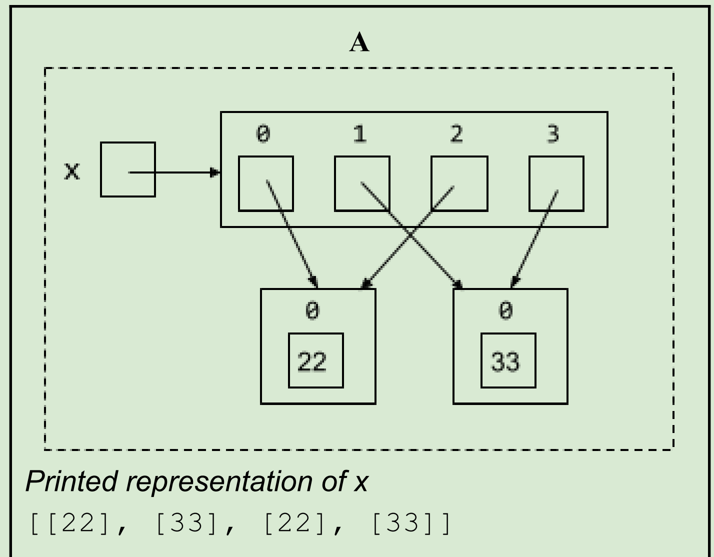
```
CODE01 = 'A'
CODE02 = 'I'
CODE03 = 'F'
CODE04 = 'A'
CODE05 = 'G'

CODE06 = 'C'
CODE07 = 'A'
CODE08 = 'F'
CODE09 = 'A'
CODE10 = 'D'

CODE11 = 'A'
CODE12 = 'H'
CODE13 = 'E'
CODE14 = 'A'
CODE15 = 'J'

CODE16 = 'A'
CODE17 = 'B'
CODE18 = 'D'
CODE19 = 'A'
CODE20 = 'I'
```

Part 9b: Modifying A Memory Diagram

```
def A_to_K(L):
    '''One of *many* different correct answers'''
    L[2] = [44] # Create the singleton list [44] in L[2]
    L[3] = L[2] # Share the new list [44] in L[2] by L[3]
    L[1].insert(0, L[2]) # Add the new list [44] in the 0th slot of L[1]
    L[0].append(L[1]) # Add a new slot 1 to L[0] containing L[1]
```

Part 9c: Drawing a Modified Memory Diagram

In this subtask, you were given the following memory diagram for `x`



You were then asked to draw the **final** memory diagram that shows the structure of the list x after executing a sequence of four statements. We will show the diagram after each of the four steps even though you were only asked to draw the final one. We will also show the printed representation of `x` after each step. Note that the printed representation alone is ambiguous because it does not show sharing.