6.1000 Midterm 2

November 12, 2025

- Write your **full name and Kerberos username** below. (Your Kerberos is your MIT email without the @mit.edu. It is ***not*** your MIT ID number.)
- The exam will begin at **3:05 pm** and end at **4:25 pm.**
- The exam is **closed-book** – no notes, electronics, or additional resources are allowed.
- You may separate the sheets, but you must turn in all of them at the end.
- If you need more space, you may use any of the blank pages. If you need additional space beyond those, ask a staff member to bring you some scratch paper, and turn it in at the end. Write your name and Kerberos on the scratch paper as well. You may not use your own scratch paper.

- There are four questions total. They will be weighted roughly equally. Make sure you spend enough time on each question.
- For questions that ask you to write Python code, do your best on syntax, and **focus on expressing the computational ideas.** Small syntax errors may be penalized lightly or not at all.
- If you are unsure about certain Python details, write down your assumptions, and do your best to make progress.
- For questions that ask you to **explain** an outcome or result, provide a **brief justification** of your reasoning. An answer alone without justification will receive no credit.

Name: **SOLUTIONS** _____

Kerberos: _____

BLANK PAGE

Reference list of some built-in Python functions

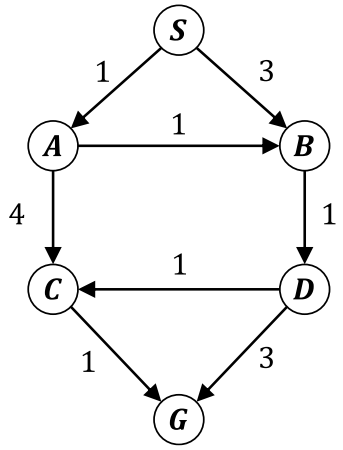
```
str.count(substr)
str.find(substr)
str.strip(chars=None)
str.split(separator=None)
str.join(iterable)

list.count(value)
list.index(value)
list.append(value)
list.extend(iterable)
list.remove(value)
list.insert(index, value)
list.pop(index=-1)
list.reverse()
list.sort(reverse=False)
list.clear()
list.copy()

dict.keys()
dict.values()
dict.items()
dict.get(key, default=None)
dict.update(mapping)
dict.pop(key, default)
dict.clear()
dict.copy()
```

Question 1

Consider the graph shown below.



Part A
Suppose we run Dijkstra's algorithm to find the shortest path from the start S to the goal G. What path will be returned? No explanation is needed.

Solution:

$$S \rightarrow A \rightarrow B \rightarrow D \rightarrow C \rightarrow G$$

Total weight is 5.

Question 1 (continued)

Part B

Suppose we add a directed edge $S \rightarrow G$ with weight 4 and run Dijkstra's algorithm again. Louis Reasoner claims that because this edge forms the new shortest path from S to G, Dijkstra's would terminate immediately after expanding from S to G.

Explain instead, after S gets expanded to its neighbors, which other nodes ***must also*** be expanded to their neighbors, **before *G* itself** gets expanded.

Solution:

A*, *B*, and *D

Dijkstra's begins by expanding S to its neighbors, putting $(1, A)$, $(3, B)$, and $(4, G)$ on the priority queue.

Next, it takes the $(1, A)$ off the queue, which expands to $(2, B)$ and $(5, C)$. When we put those on the queue, $(2, B)$ effectively overshadows $(3, B)$.

Some implementations of Dijkstra's also remove $(3, B)$ from the queue at this point, because $(2, B)$ will always be expanded before $(3, B)$. For simplicity, we will assume $(3, B)$ is removed.

Now $(2, B)$ is the smallest element on the queue, and that expands to $(3, D)$.

Then $(3, D)$ is expanded to $(4, C)$ and $(6, G)$.

At this point, $(4, G)$ and $(4, C)$ are the smallest elements on the queue. A valid implementation of Dijkstra's could pick either to expand next. Hence, C is not guaranteed to be expanded before G, but our analysis above shows A, B, and D are.

Common mistakes:

- Expanding B before expanding A, and therefore using an incorrect shortest path distance for B of 3 rather than 2.
- Identifying that C and G are both in the priority queue with distances of 4, but incorrectly claiming that Dijkstra's is required to expand C before expanding G.

Question 1 (continued)

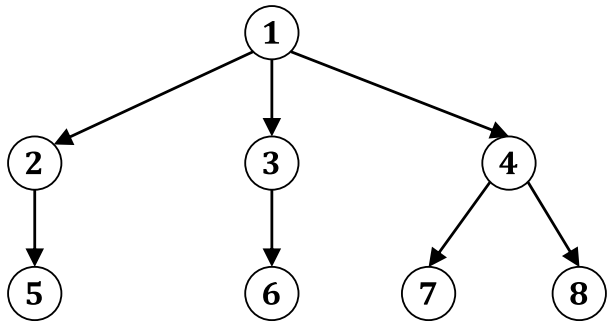
Part C

Consider the following code, which uses a queue-based implementation for breadth-first search and depth-first search, except it alternates between the two strategies for choosing which node to expand. For simplicity, we only consider graphs that are **directed trees.**

```
def alternate_bfs_dfs(tree, root):
    queue = [root]
    counter = 0
    while len(queue) > 0:
        if counter % 2 == 0:
            current_node = queue.pop(0)
        else:
            current_node = queue.pop(-1)
        counter += 1

        print(current_node)
        for next_node in neighbors(tree, current_node):
            queue.append(next_node)
```

Given the tree below, suppose that the **`neighbors()`** function returns a list of a node's children in **left-to-right order.** If **`alternate_bfs_dfs()`** is called on this tree with node 1 as the root, determine the order in which nodes are printed.



(Write your answer on the following page.)

Question 1 (continued)

Part C (continued)
Write your answer down the right column in the table below. Justify by showing the state of the queue at each loop iteration. (You do not have to use all the rows.)

Queue state	Node printed
[1]	1
[2, 3, 4]	4
[2, 3, 7, 8]	2
[3, 7, 8, 5]	5
[3, 7, 8]	3
[7, 8, 6]	6
[7, 8]	7
[8]	8

Common mistakes:

- Traversing the graph only using BFS or DFS instead of tracing through the code.
- Replacing the entire queue with the popped node's neighbors, rather than just removing the one popped node and appending neighbors to the end.

Question 2

Consider the following code.

```
 1      def avg(numbers):
 2          total = 0
 3          count = 0
 4          while count < 3:
 5              total += numbers[count]
 6              count += 1
 7          return total / count
 8
 9      def test_avg(seq_length):
10          sequence = list(range(seq_length))
11          expected = (seq_length - 1) / 2
12          assert avg(sequence) == expected
13
14      def run_tests():
15          results = []
16          for n in (3, 2):
17              try:
18                  test_avg(n)
19              except IndexError:
20                  results.append(False)
21              else:
22                  results.append(True)
23          return results
24
25      run_tests()
```

When this code is run, line 20 is executed exactly once due to an **`IndexError`.** On the following page, draw an environment diagram showing the frames and objects in memory ***at the moment when the*** **`IndexError`** ***is raised.*** (This is ***not*** the same as when line 20 is executed.)

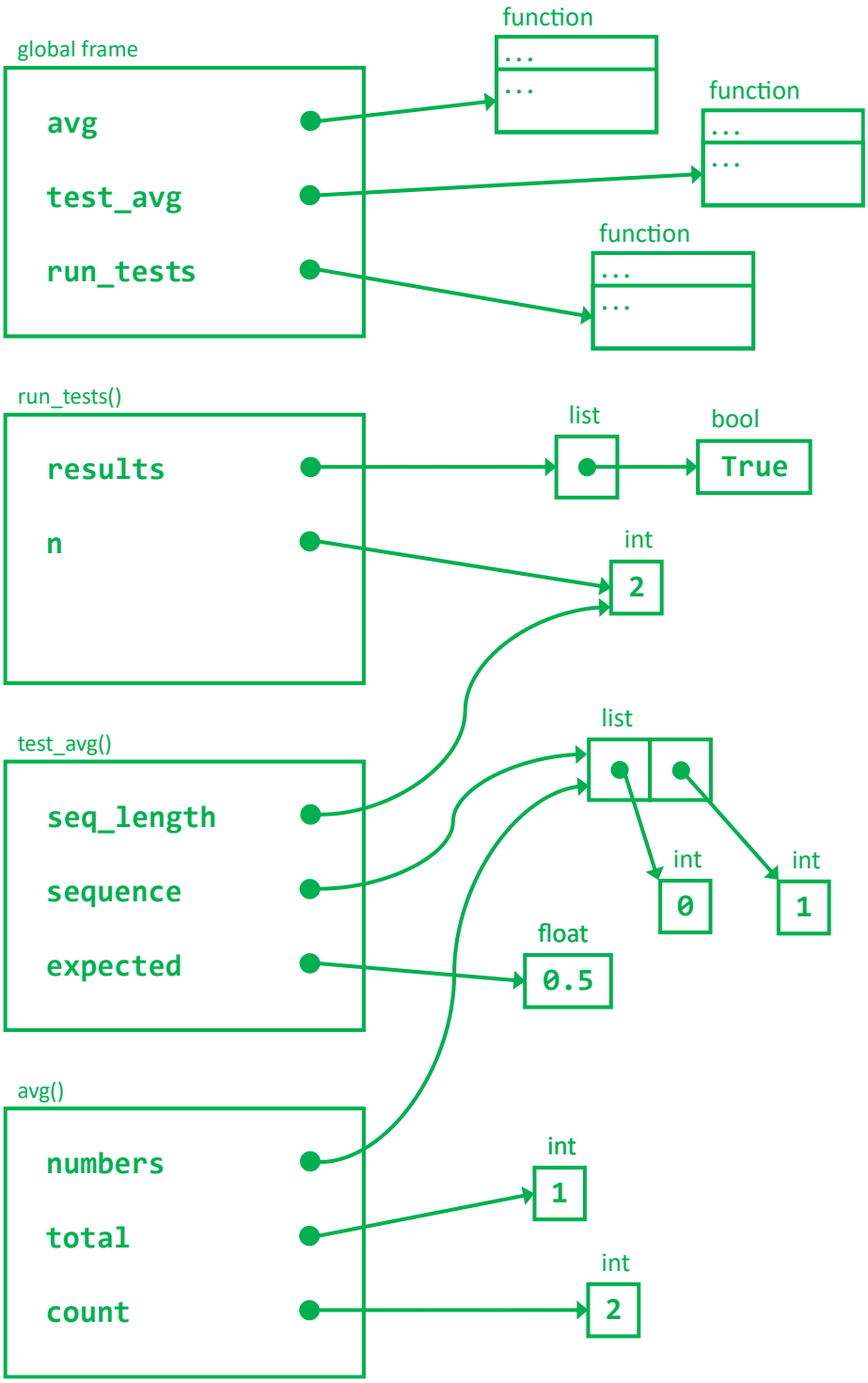
In addition to the global frame, you should only show frames for ***active*** function calls. It is fine to cross out frames for completed calls, or not to show them at all.

Label all objects with their type. *You do not need to fill in the contents of function objects.* You also do not need to show any objects that have no references to them.

Question 2 (continued)

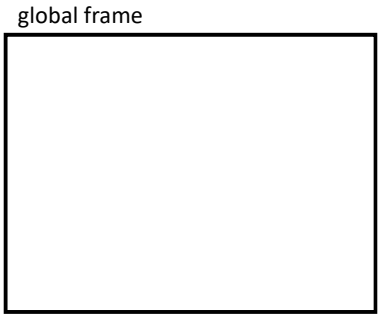
Draw your environment diagram here.

Solution:



Question 2 (continued)

Use this page only if you would like to start over on the diagram. If so, clearly indicate that on the previous page. We will grade either the previous page or this one, but not both.



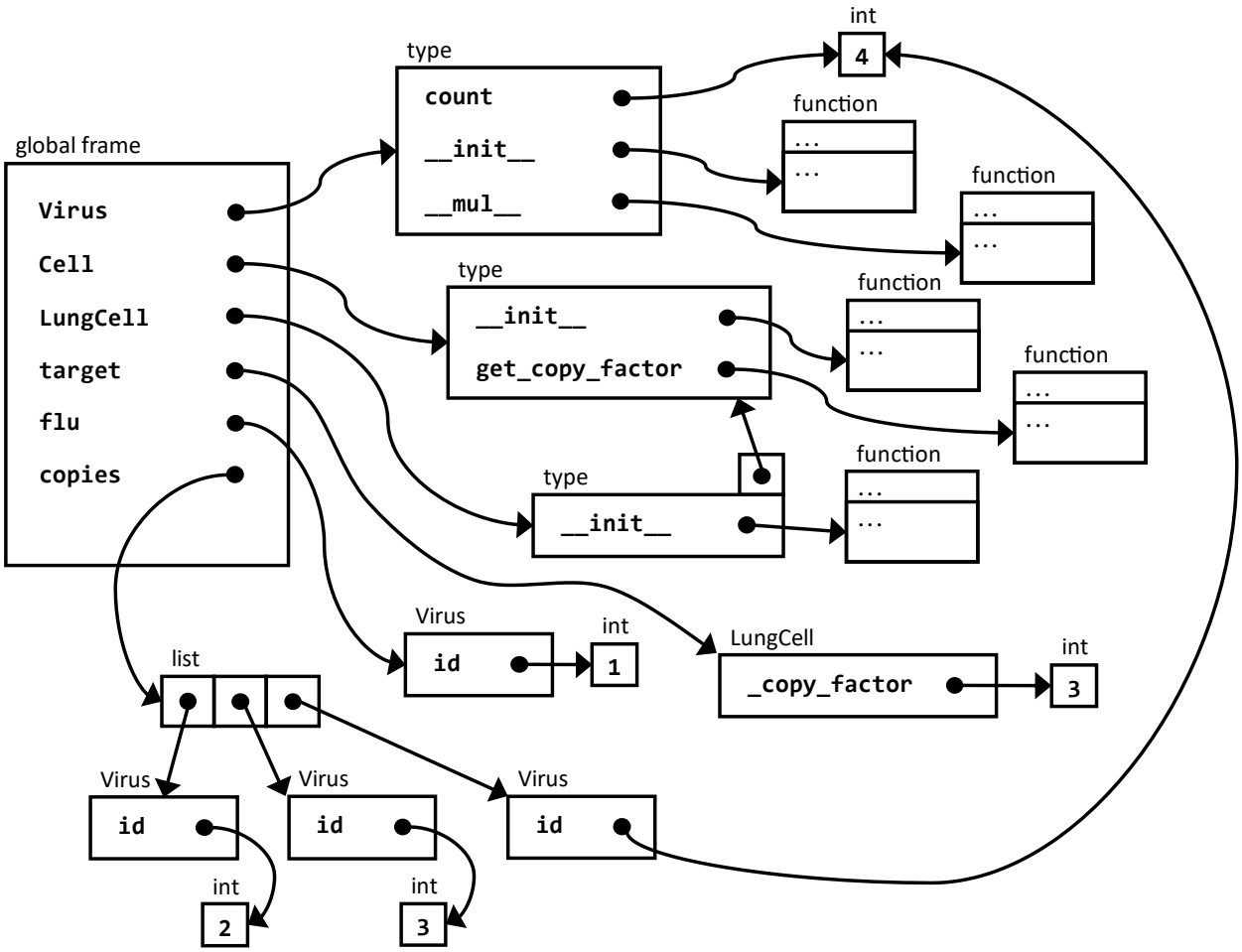
Common mistakes:

- Treating function objects as function call frames.
- Not removing frames for completed function calls.
- Missing `n` variable in `run_test()` frame.

BLANK PAGE

Question 3

Below is an environment diagram showing the state of memory after running some code. (Note that the function objects are not empty; we've just omitted their contents.)



Unfortunately, we've lost the code that produced this environment, but we know the last three lines were:

```
flu = Virus()
target = LungCell()
copies = flu * target
```

(continued on next page)

Question 3 (continued)

Given the diagram and the code fragment on the previous page, reconstruct the class definitions for **`Virus`**, **`Cell`**, and **`LungCell`** preceding the code. (Use the following page as additional space for your answer if needed.)

For full credit, your answer needs to have the following features:

- No direct access to the **`_copy_factor`** attribute outside of the class that defines it.
- The **`__init__()`** methods' parameter lists should be consistent with how the **`flu`** and **`target`** objects were created.

Your code may create additional objects in memory as long as there are no references to them after executing your code followed by the code fragment above.

```
# your code here
```

(solution on next page)

Question 3 (continued)

(Use this page as additional space for your answer if needed.)

`# your code here`

Solution:

```
class Virus:

    count = 0

    def __init__(self):
        Virus.count += 1
        self.id = Virus.count

    def __mul__(self, cell):
        return [Virus() for _ in range(cell.get_copy_factor())]

class Cell:

    def __init__(self, copy_factor):
        self._copy_factor = copy_factor

    def get_copy_factor(self):
        return self._copy_factor

class LungCell(Cell):

    def __init__(self):
        super().__init__(copy_factor=3)
```

Common mistakes:

- Defining **`Virus.count`** as a function. The name needs to be assigned to an integer.
- Incomplete relationship between **`Virus.count`** and **`self.id`.**
- Too many parameters in the **`__init__()`** methods.
- Using **`Virus.__init__()`** instead of **`Virus()`** to create a new **`Virus`** object.
- Incorrect syntax for class **`LungCell`** to inherit from class **`Cell`.**
- Redefining **`_copy_factor`** as an attribute of **`LungCell`** instances (e.g., assigning **`self._copy_factor`** within **`LungCell.__init__()`**).
- Incorrect syntax or placement of **`super().__init__()`** call.

BLANK PAGE

Question 4

In Problem Set 4, we ran simulations of MBTA trains running on a circular track for a specified number of time steps. The output of each simulation was a list of lists we'll call **`history`**, where **`history[t][i]`** was the location of train **`i`** on the track at time step **`t`.** By running many trials of these simulations, we estimated the expected duration between a train leaving a station and the next train arriving.

Suppose we now wish to estimate, ***for a given passenger,*** **the expected travel time between stations,** meaning: from when they walk into a station, to when they get off the train at a later station. This includes the time spent waiting at the first station for a train to arrive.

To model this, we can leverage the **`history`** data from our MBTA train simulations. Given the history output from a single simulation, write the following function to determine how many time steps it would take a passenger to travel between stations at **`start_loc`** and **`end_loc`,** assuming they walked into the first station at time step **`start_time`.**

```
def get_travel_time(history, start_loc, end_loc, start_time):
    """
    Determine how long a passenger's trip on the MBTA would take.

    Parameters:
        history (list): A history of train locations at each time step.
            history[t][i] is the location of train i at time step t.
        start_loc (int): The location of the passenger's start station.
        end_loc (int): The location of the passenger's destination station.
        start_time (int): The time step when the passenger walks into
            the station at start_loc.

    Return an int indicating the number of time steps since start_time
    until the passenger reaches the station at end_loc.
    """
```

Write your code on the following page. You may assume the following:

- All locations of stations on the train track are integers.
- Trains do not overlap or pass each other.
- The passenger gets on the first train that is available to them.
- **`history`** is long enough for the passenger to complete their trip, given their **`start_time`.**

Question 4 (continued)

Write your implementation below. You are welcome, though not required, to write helper functions.

Hint: How would you determine whether a passenger has boarded a train or not?

```
def get_travel_time(history, start_loc, end_loc, start_time):
    # your code here
```

Solution:

```
    boarded = False
    for step in range(start_time, len(history)):
        train_locations = history[step]
        if not boarded:
            for train_id in range(len(train_locations)):
                if train_locations[train_id] == start_loc:
                    boarded = True
                    break
        else:
            if train_locations[train_id] == end_loc:
                return step - start_time
```

Common mistakes:

- Interpreting **`history[t][i]`** as a point in time rather than a location/position along the track.
- Attempting to identify the train that the passenger boards by looking only at **`start_time`** for train locations that are less than **`start_loc`.** Because the track is a loop, this misses the case where the next train to arrive at **`start_loc`** is "in front" of **`start_loc`,** but will loop back around, past the point where location values jump back to zero.
- Need to break or return immediately after the train first reaches **`end_loc`.** Otherwise, the train may complete another full loop around the track in **`history`.**

BLANK PAGE

BLANK PAGE

BLANK PAGE