6.1000 Midterm 1

October 6, 2025

- Write your **full name and Kerberos username** (your MIT email without the @mit.edu) below.
- 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.

- 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: _____

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 following code:

```
word = "eagerbeaver"
result = []
for i in range(len(word)):
    result = result + [len(word.split(word[i]))]
```

Part A
After running this code, what would be the output of **`print(result)`?**

Part B
How many **`list`** objects are created when running this code? Explain briefly.

Question 1 (continued)

Part C

Rewrite the code on the previous page so that: a) it avoids using the built-in **`len()`** and **`str.split()`** functions, and b) it creates only a single **`list`** object throughout its execution. After running your code, the variable **`result`** should refer to an equivalent (**`==`**) object as before.

Hint: You may use other **`str`** functions.

Question 2

Given a Python **`list`** of unique numbers, consider the task of locating the index positions of the largest **`k`** numbers and returning a **`list`** of those indices in *increasing order of index value.* The input list should *not* be mutated, and you may assume **`k`** does not exceed the length of the list.

Part A
Alyssa P. Hacker proposes the following code to achieve the task. Unfortunately, there are a few problems with it.

```
1    def locate_top(numbers, k):
2        numbers_indexed = []
3        for i in range(len(numbers)):
4            numbers_indexed.append((numbers[i], i))
5        numbers_indexed.sort()
6        indices = []
7        for (num, i) in numbers_indexed:
8            indices.append(i)
9        return indices
```

(continued on next page)

Question 2 (continued)

Part A (continued)
Below, write what the expression `locate_top([5, 1, 4, 2, 3], 3)` would return for Alyssa's code. If it would raise an error instead of returning, say the line number where the error occurs and briefly explain why.

Then, writing directly on the previous page's code, change no more than five lines to correct the implementation. (We will count a single line change as either editing a line, crossing one out, or inserting a new one.)

Question 2 (continued)

Part B

Ben Bitdiddle has a different approach to solving the problem. His code is below. Unfortunately, it is also incorrect.

```
1      def locate_top(numbers, k):
2          indices = []
3          remaining = numbers
4          for _ in range(k):
5              num = max(numbers)
6              loc = numbers.index(num)
7              remaining.remove(loc)
8              indices.append(loc)
9          return indices
```

(continued on next page)

Question 2 (continued)

Part B (continued)
As in Part A, write below what the expression **`locate_top([5, 1, 4, 2, 3], 3)`** would return for Ben's code. If it would raise an error instead of returning, say the line number where the error occurs and briefly explain why.

Then, writing directly on the previous page's code, change no more than five lines to correct the implementation.

Question 3

Consider the following code:

```
1      def classifies_as(group, target):
2          if group == target:
3              return True
4          for child in taxonomy.get(group, []):
5              if classifies_as(child, target):
6                  return True
7          return False
8
9
10     taxonomy = {
11         "canine": ["dog", "wolf", "fox"],
12         "dog": ["beagle", "husky"],
13     }
14     classifies_as("canine", "wolf")
```

Part A
This code runs without errors. How many calls to **`classifies_as()`** are made in total? Explain briefly.

Question 3 (continued)

Here is the code reproduced:

```
1      def classifies_as(group, target):
2          if group == target:
3              return True
4          for child in taxonomy.get(group, []):
5              if classifies_as(child, target):
6                  return True
7          return False
8
9
10     taxonomy = {
11         "canine": ["dog", "wolf", "fox"],
12         "dog": ["beagle", "husky"],
13     }
14     classifies_as("canine", "wolf")
```

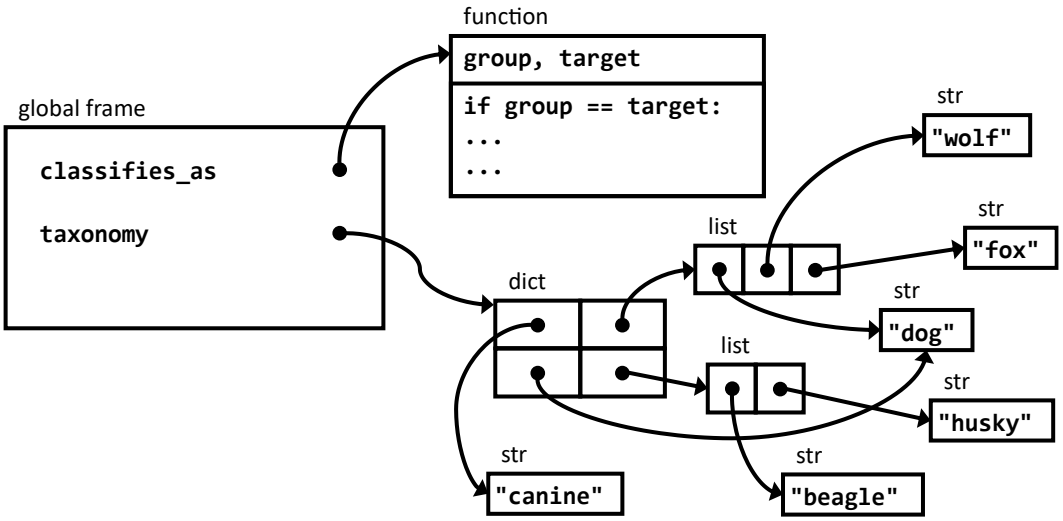
Part B

On the following page, we have started a diagram showing the state of memory right before line 14 is executed. Extend the diagram to show the state of memory *during* the execution of line 14 and ***right before*** line 3 is executed.

You may add any function call frames and objects as needed. However, you do not need to show any objects that have no references to them.

Question 3 (continued)

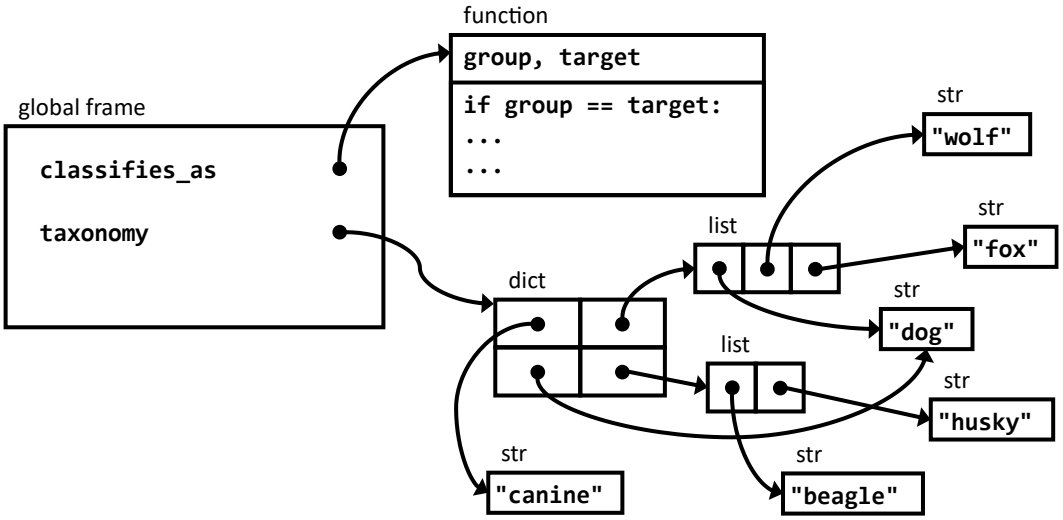
Part B (continued)



Question 3 (continued)

Part B (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.



Question 4

Recall that in Problem Set 2, we were searching for a model $\boldsymbol{y = ax + b}$ that describes the relationship between the amount of dust on a solar panel (x) and its power efficiency (y). Given a set of x and y data points, we could determine the mean squared-error (MSE) for any setting of the parameters a and b. We also noted the MSE is a quadratic expression in terms of a and b, and when we freeze the value of a, the MSE becomes a quadratic expression $\boldsymbol{\alpha b^2 + \beta b + \gamma}$ in terms of a single variable b.

We then provided the function **`slice_mse_model()`**, which computes $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, **and** $\boldsymbol{\gamma}$ for a **given setting of** $\boldsymbol{a}$**.** However, note that the implementation considers all the x and y data points each time it is called. For example, the individual y-values get squared and summed into **`gamma`.** This can be wasteful when the function is called repeatedly.

```
def slice_mse_model(x_vals, y_vals, a):
    assert len(x_vals) == len(y_vals)
    num_points = len(x_vals)
    alpha = 0
    beta = 0
    gamma = 0
    for i in range(num_points):
        x, y = x_vals[i], y_vals[i]
        alpha += 1
        beta += 2*a*x - 2*y
        gamma += y**2 + (a*x)**2 - 2*y*a*x
    alpha /= num_points
    beta /= num_points
    gamma /= num_points
    return [alpha, beta, gamma]
```

(continued on next page)

Question 4 (continued)

Consider a more efficient strategy where we precompute some values based on the data points, so we can use them directly in **`slice_mse_model()`**. The **`data_sums()`** function below constructs a **`dict`** with such precomputed values.

On the following page, implement a revised **`slice_mse_model()`**, where in place of the full data points, it accepts a precomputed **`dict`** output from **`data_sums()`** and the number of data points it was computed from.

```
def pairwise_mul(values1, values2):
    assert len(values1) == len(values2)
    num_points = len(values1)
    result = []
    for i in range(num_points):
        result.append(values1[i] * values2[i])
    return result


def data_sums(x_vals, y_vals):
    return {
        "x": sum(x_vals),
        "y": sum(y_vals),
        "x2": sum(pairwise_mul(x_vals, x_vals)),
        "xy": sum(pairwise_mul(x_vals, y_vals)),
        "y2": sum(pairwise_mul(y_vals, y_vals)),
    }
```

(continued on next page)

Question 4 (continued)

```
def slice_mse_model(accum, num_points, a):
    """
    For a linear regression model, determine the MSE quadratic
    polynomial over the parameter "b", when the parameter "a" is
    fixed at a given value.

    Parameters:
        accum (dict): The output of data_sums().
        num_points (int): The number of (x, y) data points the
            values in accum were computed from.
        a (float): The "a" parameter in the linear model y = ax + b.

    Return a list of float coefficients [alpha, beta, gamma],
    specifying the desired quadratic polynomial (alpha * b**2
    + beta * b + gamma) in terms of "b".
    """
    # your code here
```

BLANK PAGE

BLANK PAGE

BLANK PAGE

BLANK PAGE