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.

SOLUTIONS

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)`?**

Solution:

`[5, 3, 2,  5, 3, 2,  5, 3, 2,  5, 3]`

Common mistakes:

- When **`.split()`** is called with a separator that matches the beginning or end, an empty string **`""`** is included in the list it returns. (Why does this make sense?)

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

Solution:

34 list objects

- 1 empty list initially created
- 3 lists created in each loop iteration, 11 iterations total
  - output of **`word.split()`**
  - list with single **`int`** object
  - concatenation with existing **`result`** creates new list

Common mistakes:

- The **+** operator on two lists creates a new list.
- Need to count the initial empty list.
- Need to provide an explanation.

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.

Solution:

```
word = "eagerbeaver"
result = []
for char in word:
    result.append(word.count(char) + 1)
```

Explanation (not required):

`word.split()` removes instances of the separator and collects the substrings in between and outside. The number of such substrings is thus the number of separator occurrences plus one. We avoid creating new lists by mutating the original empty list.

Common mistakes:

- Wrapping an object in square brackets **`[obj]`** creates a list.
- Incorrect Part A's were taken into account when grading Part C.

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()
         numbers_indexed.reverse()
6        indices = []
7        for (num, i) in numbers_indexed[:k]:
8            indices.append(i)
9        return ~~indices~~ sorted(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.)

Solution:

```
[1, 3, 4, 2, 0]
```

Explanation (not required):

Alyssa's code sorts the numbers in ascending order, with original indices attached. Then it reads off those indices in that order.

Common mistakes:

- No error on line 4: it successfully appends a `tuple` object.
- No error on line 5: it is possible to `.sort()` a list of tuples.
  - If two items can be compared with `<`, then they can be sorted into a sequence.
- No error on line 7: `(num, i)` is a valid way to assign two loop variables.
  - See Lecture 7 code.

Common mistakes when correcting code:

- If getting the first `k` elements from `numbers_indexed`, need to `.reverse()` the list first.
- Need to sort index values before returning.
- `indices.sort()` mutates the list but evaluates to `None`, so shouldn't return that expression directly.

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.copy()
4          for _ in range(k):
5              num = max(numbers) max(remaining)
6              loc = numbers.index(num)
7              remaining.remove(loc) remove(num)
8              indices.append(loc)
9          return indices sorted(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.

Solution:

Error on line 7.

On the first iteration through the loop, the max number is 5 at index 0. When we try to remove 0, Python can't find 0 in the list and raises a `ValueError`. (We did not require you to specify the exact type of error.)

Further explanation (not required):

Ben's code attempts to repeatedly find the max of the `numbers` list and remove them from `remaining`. Problems with his code include: First, `numbers` and `remaining` are aliases of the same list. Second, he's using the index location from line 6 as the argument to `.remove()` in line 7. However, `.remove()` accepts an element value to remove, not an index location.

Common mistakes:

- No error on line 4: possible to use `_` as a variable name.
  - See Lecture 6 code. The underscore is a convention for an unused loop variable.
- Saying abstractly that `.remove()` should accept an element value instead of an index, but not specifying the argument to `.remove()` that causes it to fail.

Common mistakes when correcting code:

- Line 6 looks up index in original `numbers` list, which should not be mutated. Thus, need `remaining` to be a copy, not an alias, of `numbers`.
- Need to sort index values before returning, and make sure not to return `None`.

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.

Solution:

5 calls

This code performs a depth-first search through the **`taxonomy`** graph. It explores the **`"dog"`** subtree before continuing to **`"wolf"`** and then stops the search.

```
classifies_as("canine", "wolf")
    classifies_as("dog", "wolf")
        classifies_as("beagle", "wolf")
        classifies_as("husky", "wolf")
    classifies_as("wolf", "wolf")
```

Common mistakes:

- Need to explore the entire **`"dog"`** subtree before moving on to **`"wolf"`**.
- After finding **`"wolf"`**, this code does not continue to explore **`"fox"`**.

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.

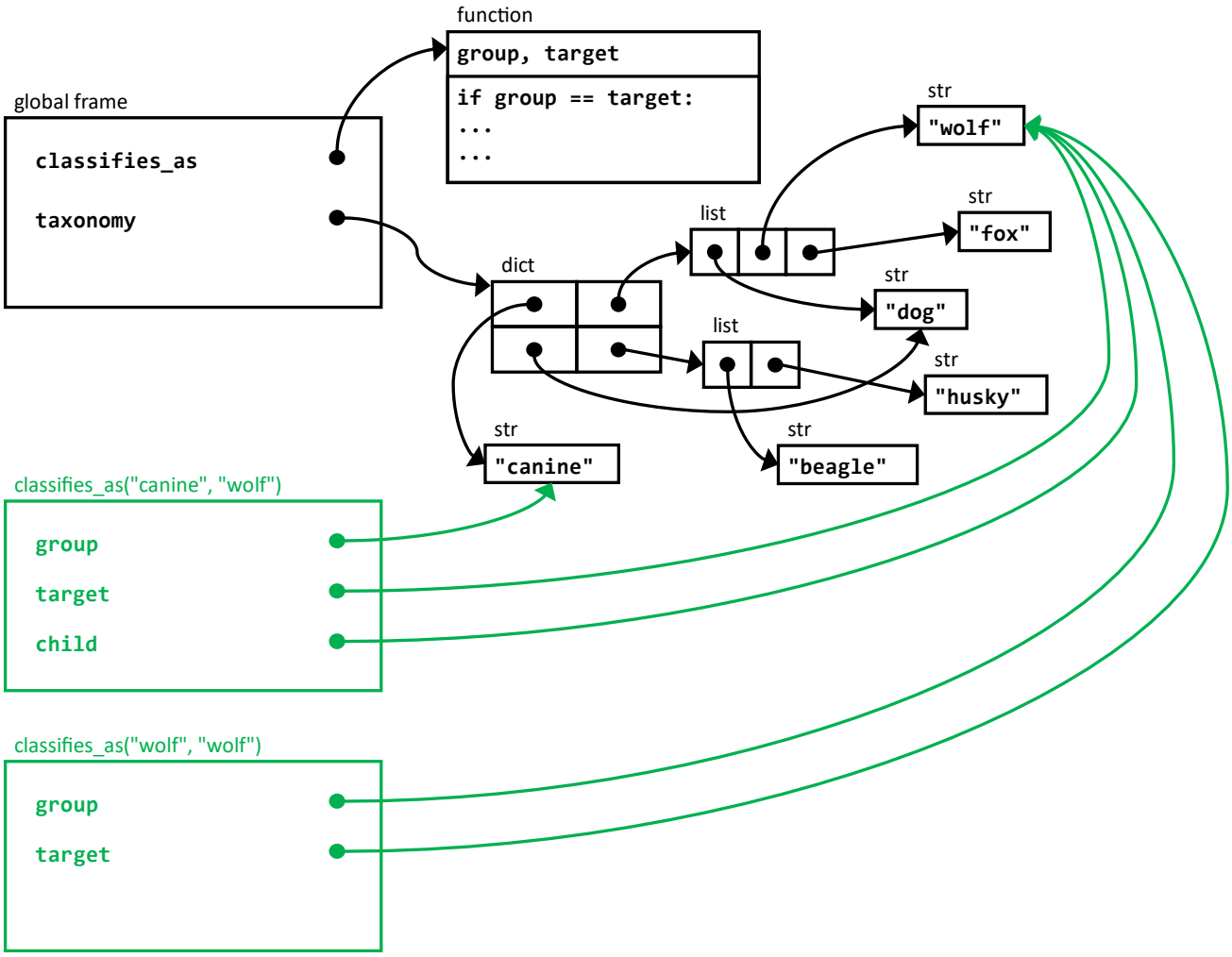
You may add any function call frames and objects as needed. However, you do not need to show any objects that have no references to them.

Common mistakes:

- Need to draw function call frames, not additional function objects.
- Inside function call frames, there should only be variable names pointing to objects outside. `str` objects do not belong inside frames.
- Frames for function calls that have completed should be removed.
- "***right before*** line 3 is executed" means when we've determined the `if` condition on line 2 is `True`, not whenever we execute line 2.

Question 3 (continued)

Part B (continued)



Explanation (not required):

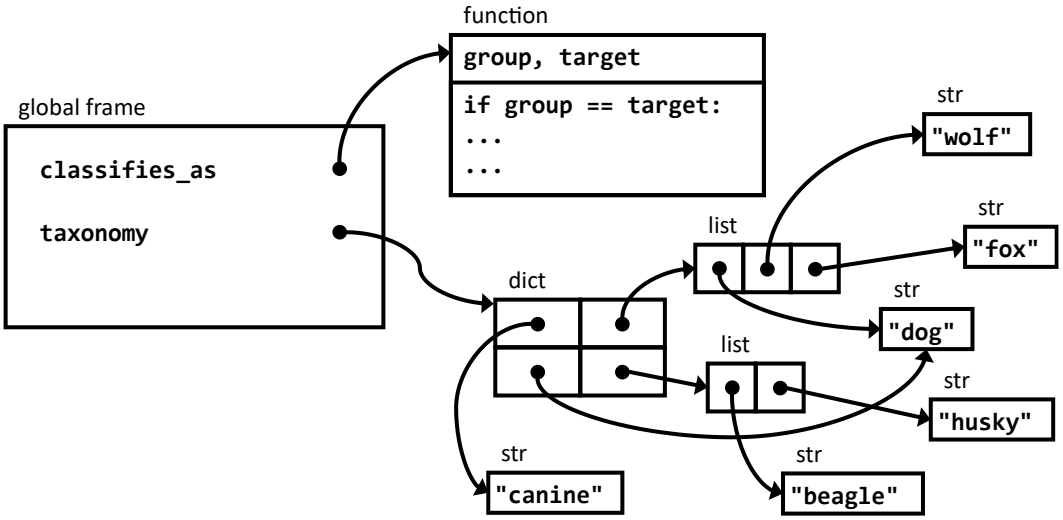
Active function call frames are stacked underneath (or "on top of") the global frame. Each function call initializes local variables **`group`** and **`target`** according to what arguments are passed into the call. When line 4 is reached, the **`for`** loop sets a **`child`** variable as well to iterate over the elements of a list.

The DFS search structure of the code results in line 3 executing exactly once when we reach the recursive call where group is assigned to **`"wolf"`**. Right before we execute that **`return`** statement, the active frames represent the top-level call and the recursive call we're about to return from.

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

    beta = (2 * a * accum["x"] - 2 * accum["y"]) / num_points

    gamma = (

        accum["y2"]

        + a**2 * accum["x2"]

        - 2 * a * accum["xy"]

    ) / num_points

    return [1, beta, gamma]
```

Common mistakes:

- **`x_vals`** and **`y_vals`** are not available anymore, should not reference them.
- No need for any loops.
- The sum of all **`x**2`** values is **`accum["x2"]`**, not **`accum["x"]**2`**.

Explanation (not required):

In the original implementation, for a term like **`2*a*x`**, the **`a`** value can be factored out of the **`for`** loop, and what remains is twice the sum of all the **`x`** values. The **`data_sums()`** function precomputes the relevant sums, which we can retrieve using **`dict`** lookup, and thus avoid looping over all x and y data points again.

BLANK PAGE

BLANK PAGE

BLANK PAGE

BLANK PAGE