6.101 Final Exam

Fall 2023

Name: **Answers**

Kerberos/Athena Username:

6 questions

3 hours

- Please **WAIT** until we tell you to begin.
- Write your name and kerberos **ONLY** on the front page.
- This exam is closed-book, and you may not use any electronic devices (including computers, calculators, phones, etc.).
- If you have questions, please **come to us at the front** to ask them.
- Enter all answers in the boxes provided. Work on other pages with QR codes may be taken into account when assigning partial credit. **Please do not write on the QR codes.**
- If you finish the exam more than 10 minutes before the end time, please quietly bring your exam to us at the front of the room. If you finish within 10 minutes of the end time, please remain seated so as not to disturb those who are still finishing their exams.
- You may not discuss the details of the exam with anyone other than course staff until final exam grades have been assigned and released.

Worksheet (intentionally blank)

1 Singing in the Rain

Intrigued by apparent changes in the climate over recent years, you've installed an automated rain gauge outside your room to keep track of daily precipitation amounts. The rain gauge reports its readings to you as a list of numbers (one per day), each indicating the amount of rain that fell in the last day.

You would like to average these numbers together to get a sense for long-term trends, so you decide to write a Python program to do so. However, as with any real-world measurement, sometimes things can go wrong. As such, there are two kinds of issues that can occur with your rain gauge:

- It can report negative numbers. These are obviously erroneous (negative rainfall can't happen!) and so each negative number should be ignored when computing your average.
- Something could cause the sensor to break, in which case it will instead report the string `"ERROR"`. In this case, values after that point can't be trusted, so *every value after the error* should be ignored when computing your average.

After ignoring the erroneous values, your function should return the average of those values that remain. If the average cannot be computed, your function should return `None` instead of raising an exception.

What follows are two attempts to solve this problem. For each, you are asked to provide four separate examples of **valid**, **nonempty** inputs to the function (i.e., lists containing only numbers or `"ERROR"` and containing at least one such value) that produce a variety of results:

- one example of an input for which the function returns the correct result,
- one example of an input for which the function runs to completion but returns an incorrect result,
- one example of an input that causes an infinite loop, and
- one example of an input that causes an exception to be raised.

If any of these outcomes is not possible for a given implementation, write an `X` in the associated box instead of providing an input.

Rainfall: Attempt 1

```
def average_rainfall(rainfalls):
    total = 0
    count = 0
    ix = 0
    while rainfalls[ix] != "ERROR":
        if rainfalls[ix] <= 0:
            continue

        total += rainfalls[ix]
        count += 1
        ix += 1

    if count > 0:
        return total / count
    return None
```

Valid nonempty input that produces correct result (or X if not possible):

`[1, 2, 3, 'ERROR']`

Valid nonempty input that produces incorrect result (or X if not possible):

`X` *(no valid input)*

Valid nonempty input that results in an infinite loop: (or X if not possible)

`[1, 2, -1, 3]`

Valid nonempty input that results in an exception: (or X if not possible)

`[1, 2, 3]`

Rainfall: Attempt 2

```
def average_rainfall(rainfalls):
    out = 0
    count = len(rainfalls)
    for amount in rainfalls:
        if amount == "ERROR":
            break
        if amount >= 0:
            out += amount / count
    return out
```

Valid nonempty input that produces correct result (or X if not possible):

`[1, 2, 3]`

Valid nonempty input that produces incorrect result (or X if not possible):

`[1, 2, 3, -1]`

Valid nonempty input that results in an infinite loop: (or X if not possible)

X *(no valid input)*

Valid nonempty input that results in an exception: (or X if not possible)

X *(no valid input)*

2 Environment Diagrams

In this problem, we will consider two programs. For each, what will be printed to the screen when the program is run? If an error occurs, write all of the output up to that point as well as the approximate error message Python would produce.

For each, also draw an environment diagram showing the state of the program just before it finishes (but do not delete frames or objects even if they have been garbage collected). You do not need to rewrite the bodies of the functions in the diagram unless you want to.

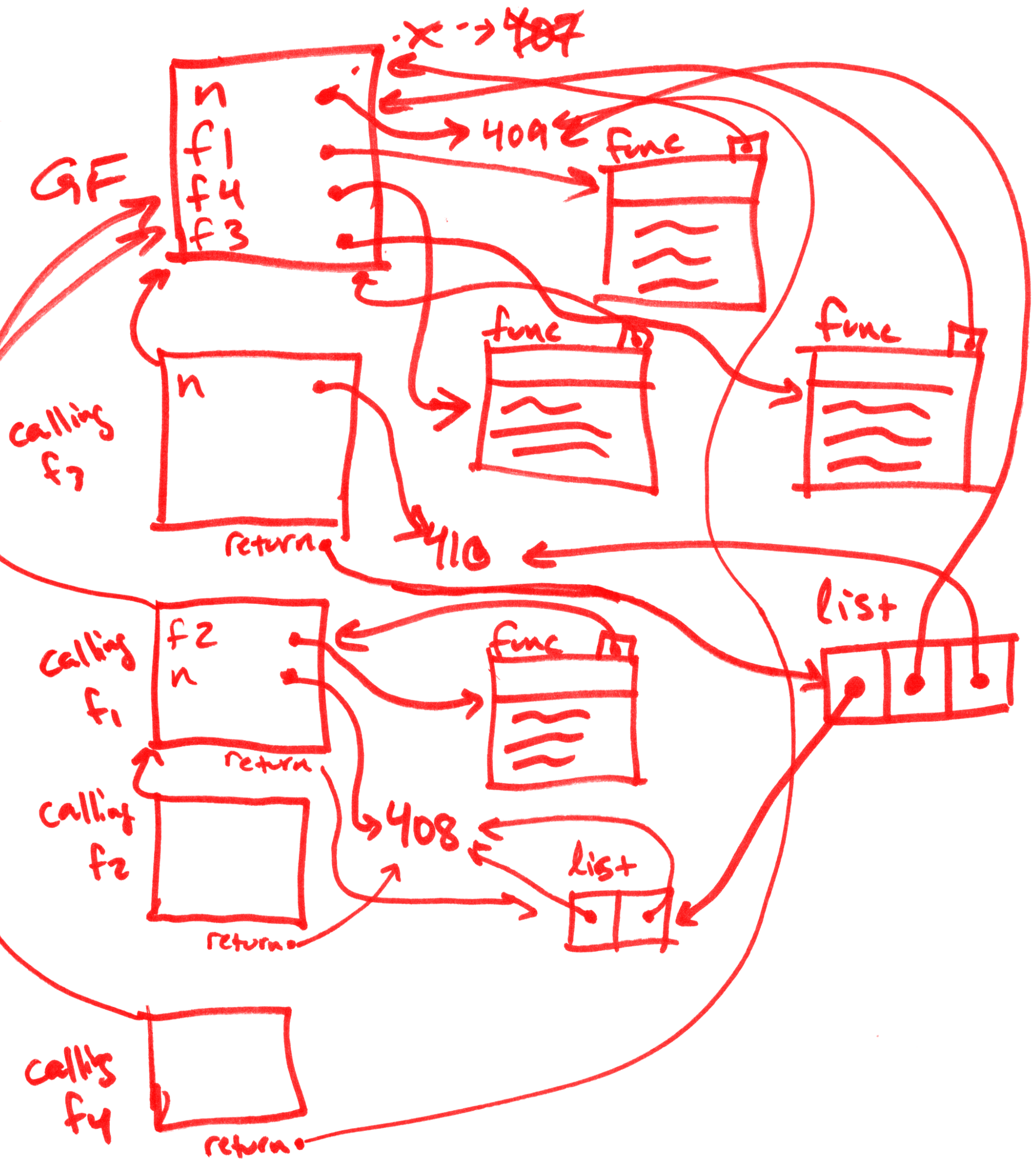
2.1 Program 1

```
n = 407

def f1():
    def f2():
        return n

    n = 408
    return [f2(), n]

def f4():
    return n

n = 409

def f3():
    n = 410
    return [f1(), f4(), n]

print(f3())
```

Program 1 Output:

```
[[408, 408], 409, 410]
```

Program 1 Environment Diagram:



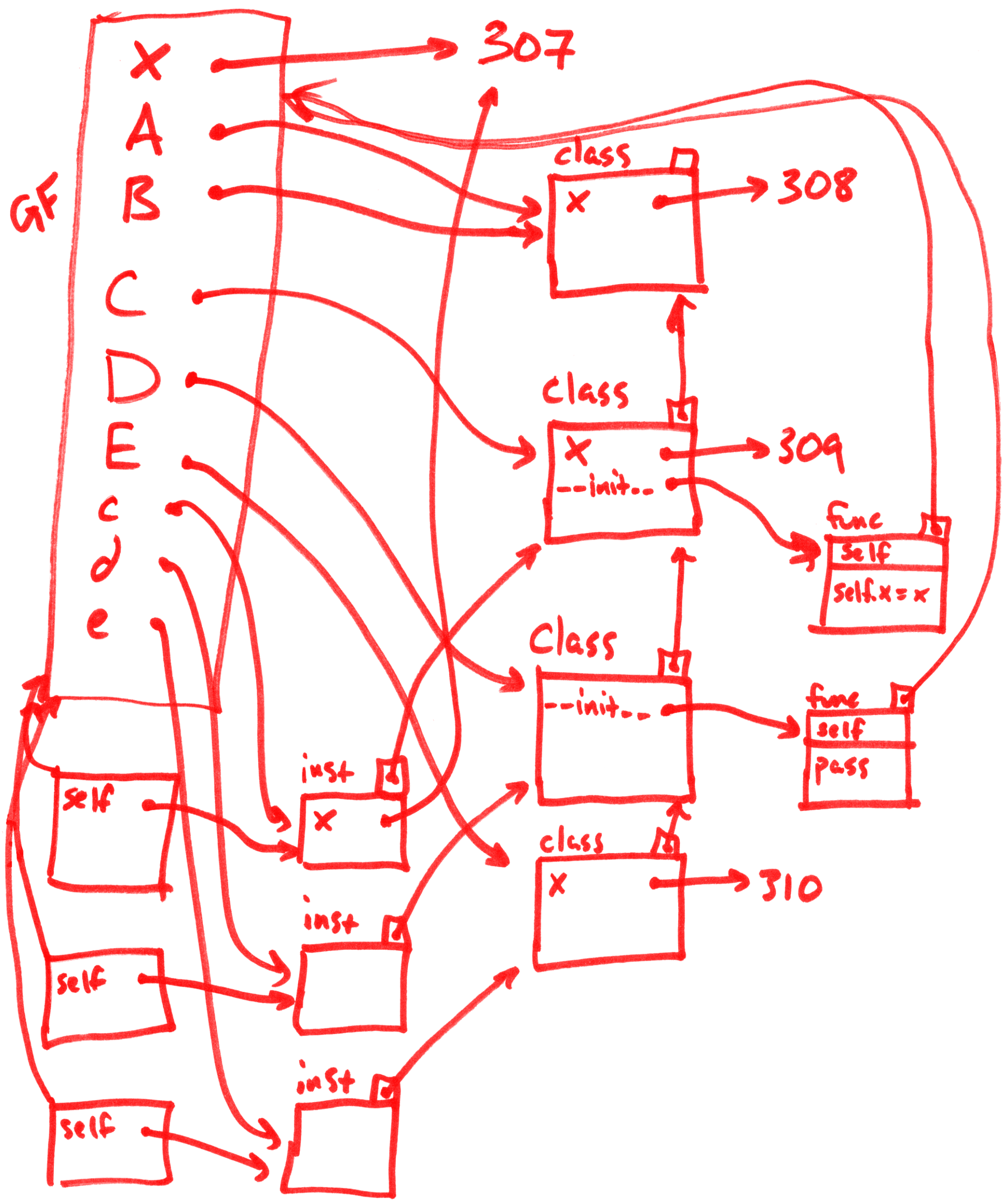
2.2 Program 2

```
x = 307

class A:
    x = 308

B = A

class C(B):
    x = 309
    def __init__(self):
        self.x = x

class D(C):
    def __init__(self):
        pass

class E(D):
    x = 310


c = C()
d = D()
e = E()
print(c.x)
print(d.x)
print(e.x)
```

Program 2 Output:

```
307
309
310
```

Program 2 Environment Diagram:



3 Translating Generators

When you arrive to work at your Macronoft internship for the summer, your supervisor Gil Bates tasks you with improving some code that was written by the last intern. Previously, Macronoft had used lists for everything internally, but customer surveys suggest that anything with the word ”generator” in it just sounds cooler and is likely to sell better, so they ask you to rewrite the old code using generators. Each of the two functions below takes a list of lists as input and produces a single list as output.

Your goal is to write a generator version of each function. Your generator should take in a a list of *generators* as input, and it should produce the same values as the list version of the code, in the same order.

For full credit, you should not convert any of the input generators into a list or tuple or similar structure but rather should work with them directly. Recall that you can loop over generators using the usual `for` loop syntax, or the built-in function `next` will return the next element from a generator (raising a `StopIteration` exception if the generator is empty).

3.1 Combining Lists

```
# List version:
def combine(lists):
    out = []
    for list in lists:
        out.extend(list)
    return out


# Generator version:
def combine_gen(gens):
```

```
    for g in gens:
        yield from g
```

3.2 Round Robin

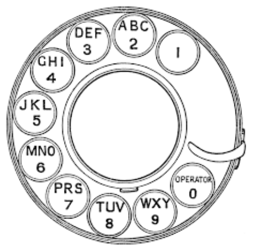
```
# List version:
def round_robin(lists):
    out = []
    for ix in range(max(len(L) for L in lists)):
        for L in lists:
            if ix < len(L):
                out.append(L[ix])
    return out


# Generator version:
def round_robin_gen(gens):
```

```
done = [False for g in gens]
while not all(done):
    for ix, g in enumerate(gens):
        if done[ix]:
            continue
        try:
            yield next(g)
        except StopIteration:
            done[ix] = True



# alternative solution:
while True:
    all_failed = True
    for g in gens:
        try:
            yield next(g)
            all_failed = False
        except StopIteration:
            continue
    if all_failed:
        return
```

Worksheet (intentionally blank)

4 New Phone, Who Dis?

In the era of rotary telephones, the telephone dial provided a bidirectional mapping between a digit and a triplet of three letters.



This mapping was often used (and sometimes still is used!) to easily communicate phone numbers. For example, ″MIT6101ROX″ corresponds to the phone number (648) 610-1769.

It can be amusing to think about the opposite problem: to figure out the words that could be formed from a given phone number. For example, ″43556″ spells out ″HELLO″.

The function `phone_words(digits)` below provides a skeleton that we can use to implement this behavior. The intention is that this function should take as input a single string containing some number of digits, and it should return a set of all of the words that could be formed from a contiguous substring of the digits in the given string (in order).

Note that you may make use of a helper function `is_word`, which takes a string as input and returns `True` if the given string represents a valid word; as well as a dictionary `key_letters`, which maps each digit to a string containing all the characters that could replace that digit in the output.

For example, the input string `"22873663"` contains the following words (among others):

- `"CAT"` from the `"228"` at the start of the string.
- `"FOOD"` from the `"3663"` at the end of the string.
- `"BAT"` from the `"228"` at the start of the string.
- `"AT"` from the `"28"` (the second and third characters).

so the output set should contain `'CAT'`, `'FOOD'`, `'BAT'`, and `'AT'` (among others).

Here is your starting code:

```
key_letters = {'0': '-', '1': '-', '2': 'ABC', '3': 'DEF', '4': 'GHI',
               '5': 'JKL', '6': 'MNO', '7': 'PQRS', '8': 'TUV', '9': 'WXYZ'}

def phone_words(digits):
    return {
        word
        for substring in digit_substrings(digits)
        for word in letters_from_digits(substring)
        if is_word(word)
    }
```

Fill in the definitions of `digit_substrings` and `letters_from_digits` on the following two pages so that this code produces a set containing all words that exist anywhere in the sequence of input digits. For full credit, your code should work for arbitrarily-long input strings, and it should work even if the specific values in the `key_letters` dictionary change.

```
def digit_substrings(digits):
    """
    Return a list, set, or generator containing all possible substrings of the
    given string of digits (including the empty string)

    >>> sorted(set(digit_substrings("1231")))
    ['', '1', '12', '123', '1231', '2', '23', '231', '3', '31']
    """
```

```
return {
    digits[s:e]
    for s in range(len(digits))
    for e in range(len(digits)+1)
}
```

```
def letters_from_digits(digits):
    """
    Return a list, set, or generator containing all possible combinations of
    letters representable by the given string of digits (regardless of whether
    they are words or not).

    >>> sorted(set(letters_from_digits("234")))
    ['ADG', 'ADH', 'ADI', 'AEG', 'AEH', 'AEI', 'AFG', 'AFH', 'AFI',
     'BDG', 'BDH', 'BDI', 'BEG', 'BEH', 'BEI', 'BFG', 'BFH', 'BFI',
     'CDG', 'CDH', 'CDI', 'CEG', 'CEH', 'CEI', 'CFG', 'CFH', 'CFI']
    """
```

```
if len(digits) == 0:
    yield ''
else:
    for letter in key_letters[digits[0]]:
        for subword in letters_from_digits(digits[1:]):
            yield letter + subword
```

Worksheet (intentionally blank)

5 Linked Dictionaries

Our Scheme interpreter from the last two labs had a lot of nice features, but one key thing that it is missing is a representation of dictionaries. However, we can implement something like dictionaries using linked lists. Here, as in the last lab, we'll use interconnected `Pair` objects to represent linked lists, and we'll represent a dictionary as a linked list where each element is a `Pair` object representing a single key→value mapping.

Recall that instances of our `Pair` class had only two instance attributes (called `car` and `cdr`, respectively), each specified at initialization time:

```
>>> p = Pair(2,3)
>>> p.car
2
>>> p.cdr
3
```

and that a linked list was represented as either

- something representing an empty linked list (here we'll use `None`), or
- a `Pair` object whose `car` is the first element in the list and whose `cdr` is a linked list containing the rest of the elements.

So, for example:

- `None` represents the empty list.
- `Pair(9, None)` represents a list with a single element (9).
- `Pair(9, Pair(8, Pair(7, None)))` represents a list containing the elements 9, 8, and 7, in that order.

For this problem, we will represent a "linked dictionary" as a linked list of the form mentioned above, but where the elements in the linked list are themselves `Pair` objects, with `car` representing a key in the dictionary and `cdr` representing the associated value.

For example, the following represents a linked dictionary that maps three numbers to their squares:

```
Pair(Pair(7, 49), Pair(Pair(8, 64), Pair(Pair(9, 81), None)))
```

In this problem, we'll ask you to write two functions that operate on linked dictionaries of this form (see the following pages for descriptions). For full credit, you should implement these functions without using any looping structures (`for` or `while`).

```
def dict_get_item(d, key):
    """
    Return the value associated with the given key in the given linked
    dictionary (d).  If the key does not exist, raise a KeyError instead.
    """
```

```
    if d is None:
        raise KeyError

    if d.car.car == key:
        return d.car.cdr

    return dict_get_item(d.cdr, key)



    # Alternative approach that finds the last Pair containing this key
    # rather than stopping as soon as we've found it once (less efficient,
    # but enables the second solution of dict_set).
    found = None
    def helper(node):
        nonlocal found
        if node is None:
            return
        if node.car.car == key:
            found = node.car.cdr
        helper(node.cdr)
    helper(d)
    if found is None:
        raise KeyError
    return found
```

```
def dict_set_item(d, key, value):
    """
    Without mutating the given linked dictionary d, return a new linked
    dictionary d2 that has the given key mapped to the given value, such that
    calls to dict_get_item(d2, key) will return the given value.
    """
```

```
if d is None:
    return Pair(Pair(key, value), None)

first = d.car
if first.car == key:
    first = Pair(first.car, value)
return Pair(first, dict_set_item(d.cdr, key, value))


# Alternative approach that just appends this key/value pair to the end
# (only works with the alternative approach from dict_get)
if d is None:
    return Pair(Pair(key, value), None)

return Pair(d.car, dict_set_item(d.cdr, key, value))
```

6 Counterpoint

You've been in the lab all week gathering data, and now it's time to look at the data. In particular, as a place to start, you'd like to write a piece of code that represents a histogram (i.e., a counter that keeps track of the number of times each value appears). You decide to do this by creating a general-purpose class to count things. But it's a nice day outside and you want to spend the day in the sun, so you decide to use ChatGPT to generate the code instead. You prompt it with the following:

"It's a nice day outside so I want to spend the day out in the sun. Please write me a Python class that can keep track of the number of times an element has appeared in its input."

ChatGPT responds with:

"Certainly! Here's a simple Python program that can be used to count the occurrences of elements in a list." It then spits out the code from page 27, which you may remove from the end of this exam.

Later that day, after the sun has gone down, you try to run the code, but it is not working as expected. There are multiple correctness, efficiency, and style issues. Take a careful look at the code and **briefly** describe the issues you find, as well as a brief note as to how they might be fixed, in the boxes below and on the facing page. For full credit, correctly identify one correctness issue, one efficiency issue, one style issue, and two additional issues of any kind.

There are *many* issues with this code, among them:

- **Correctness**: `counts` is initialized as a class attribute, which means that multiple instances (each intended to track its own elements) would actually share their counts.
- **Correctness**: The `elements` method should use *yield from* instead of *yield* (it currently yields the whole list as a single value).
- **Correctness**: The `elements` method does not make any attempt to filter out duplicates.
- **Correctness**: The `count` method introduces an attribute called `count`, such that subsequent attempts to call the `count` method would fail (looking up `count` from an instance would find this number instead of the method).
- **Correctness**: The `most_common` method should *raise* the `ValueError` on line 39 instead of returning it.
- **Efficiency**: The choice of a list as our data structure means that the `most_common` and `total` methods can be quite slow (both take quadratic time in the length of the `elements` list).
- **Style/Efficiency**: The `most_common` method recomputes `self.count(element)` multiple times, where it would be preferable to compute the result once and store it in a variable instead of repeating that computation.
- **Style**: The `most_common` method contains some magic numbers (specifically the indices `[0]` and `[1]` to look up the count of the max element and the element itself. These might be better as two separate variables rather than using a tuple at all.
- **Style**: The variable name `i` in the `count` method usually implies that `i` is an index; here, however, it is used for the elements in the list instead.
- **Style**: The attribute name `counts` is bad given the current choice of data structure, since it has nothing to do with counting.

(problem continues on next page)

You later decide to rewrite the code for the `Counter` class yourself, using a dictionary (mapping distinct elements to integers representing the number of times they have appeared) instead of a list as your internal representation. Assuming the issues from the previous part were fixed, what methods would you need to reimplement in order to correctly make use of this different representation? Which could stay the same? For those that need to be reimplemented, briefly explain why they need to change. For those that can stay the same, why do they not need to change?

Does `__init__` need to change? [✓ **Yes**] / **No**
Why or why not?

`__init__` needs to change in order to initialize `counts` as an instance variable rather than as a class variable if we assume that this was already "fixed" from the previous part. If not, it can stay the same because the current implementation of `__init__` only depends on other methods in the class (not on the specific nature of `count`).

Does `count` need to change? [✓ **Yes**] / **No**
Why or why not?

This implementation would not work with the new choice of representation since dictionaries don't contain duplicate elements. Instead, it should look up the key in the new dictionary representation and return the associated value (or `0` if that key doesn't exist).

Does `elements` need to change? **Yes** / [✓ **No**]
Why or why not?

Fixing the current version would involve filtering out duplicates, which is no longer necessary. But assuming that that fix had been made, something like `yield from set(self.counts)` would work just fine with this new representation, so there is no need to make additional changes.

Does `increment` need to change? **[✓ Yes]** / **No**
Why or why not?

This works directly with the `counts` attribute and assumes it is a list. `increment` would need to change to add one to the key associated with the given element (or to add it to the dictionary if it's not already there).

Does `total` need to change? **Yes** / **[✓ No]**
Why or why not?

`total` is written entirely in terms of other methods and so wouldn't need to change assuming those were updated properly.

Does `most_common` need to change? **Yes** / **[✓ No]**
Why or why not?

`most_common` is written entirely in terms of other methods and so wouldn't need to change assuming those were updated properly.

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Code for Counter Exercise

```
class Counter:
    counts = []

    def __init__(self, elements):
        """Initialize this instance by adding each element from the input."""
        for elt in elements:
            self.increment(elt)

    def count(self, elt):
        """Return the number of times that the given element has appeared."""
        self.count = 0
        for i in self.counts:
            if i == elt:
                self.count += 1
        return self.count

    def elements(self):
        """A generator that yields all of the distinct elements (no repeats)."""
        yield self.counts

    def increment(self, elt):
        """Add one instance of the given element elt to our counter."""
        self.counts.append(elt)

    def total(self):
        """Return the total number of elements that have appeared."""
        return sum(self.count(elt) for elt in self.elements())

    def most_common(self):
        """
        Return the element that has appeared most frequently.  If no elements
        have appeared, raise a ValueError instead.
        """
        most_common = None, 0
        for element in self.elements():
            if self.count(element) > most_common[1]:
                most_common = (element, self.count(element))
        if most_common[1] == 0:
            return ValueError('no elements to count!')
        return most_common[0]
```

Worksheet (intentionally blank)