6.009 Quiz 2

Fall 2021

Name:

Kerberos/Athena Username:

5 questions 1 hour and 50 minutes

- Please **WAIT** until we tell you to begin.
- This quiz is closed-book, but you may use two 8.5 $\times$ 11 sheets of paper (both sides) as a reference.
- 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/spaces 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 quizzes.
- You may not discuss the details of the quiz with anyone other than course staff until final quiz grades have been assigned and released.

Worksheet (intentionally blank)

1 A Sense of `self`

Part a.

Below is the code for a class representing robots coming off a factory line. Each robot has a two-letter ID (a string, e.g. `"aa"`) and optionally associates itself with an owner (a string, or `None` if the robot has not been purchased). The `count` attribute should keep track of the total number of robots that have been created.

In several places throughout the code, blanks have been inserted. For each blank, indicate whether, in order for the program to work:

S:	`self.` must be inserted into the blank,
R:	`Robot.` must be inserted into the blank,
SR:	one of `self.` or `Robot.` must be inserted into the blank (but either would work), or
N:	nothing needs to be inserted into the blank (the program works if the blank is left empty).

Specify your answer by writing one of the options above (**S**, **R**, **SR**, or **N**) into each blank.

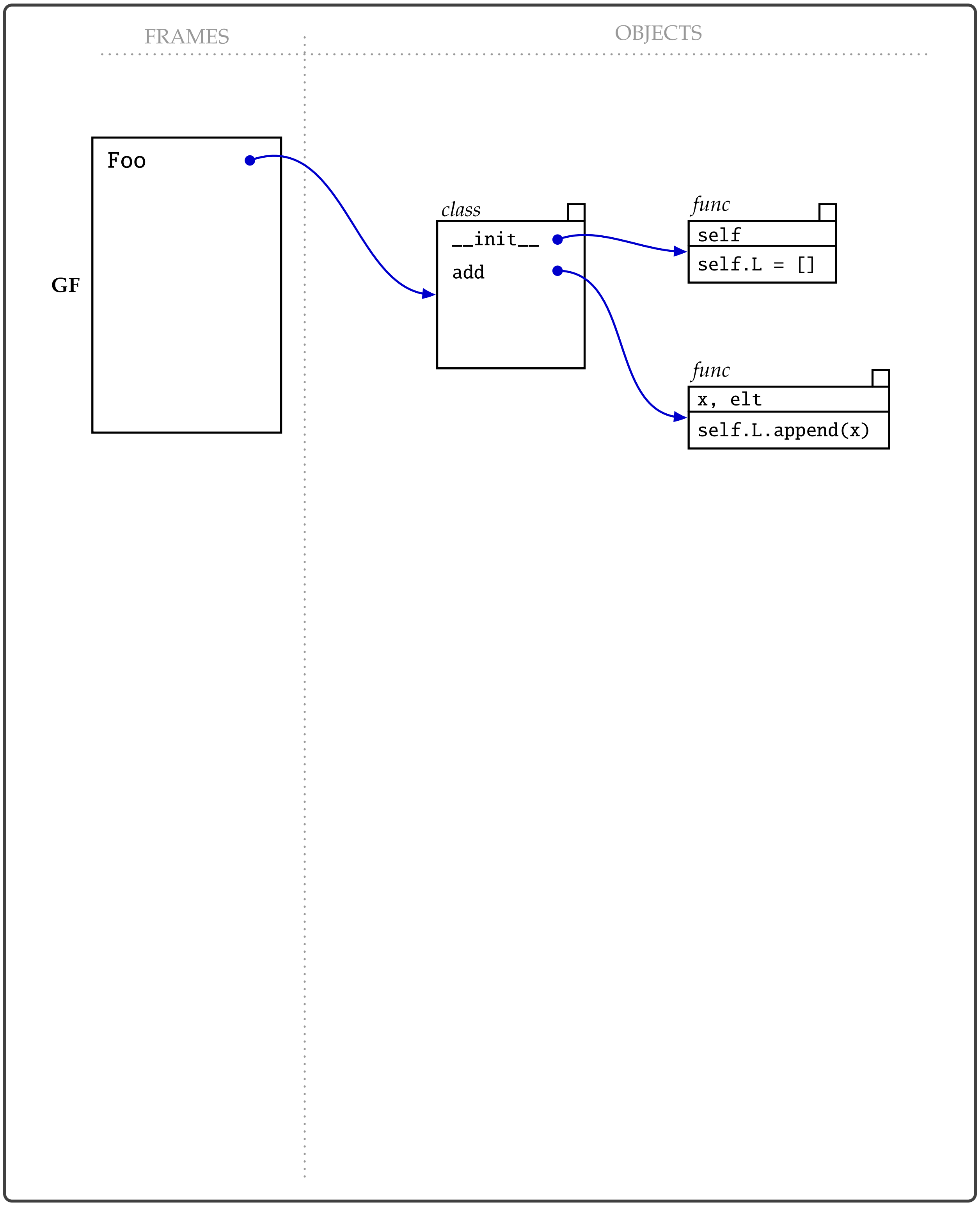
```
class Robot:
    count = 0

    def __init__(self, id_):
        _______id = _______id_
        _______count = _______count + 1
        _______owner = _______None

    def purchase(self, user):
        _______owner = _______user
        _______output_string = "Robot with id " + _______id + " purchased by " + _______user
        print(_______output_string)

    def get_information(self):
        print("This is robot " + _______id + " (one of " + str(_______count) + ")")
        if _______owner is _______None:
            _______output_string = "Not yet purchased"
        else:
            _______output_string = "Owned by " + _______owner
        print(_______output_string)
```

Part b.

Consider the following program:

```
01 |   class Foo:
02 |       def __init__(self):
03 |           self.L = []
04 |
05 |       def add(x, elt):
06 |           self.L.append(x)
07 |
08 |   f = Foo()
09 |   self = f
10 |   f2 = Foo()
11 |   self.add(7)
12 |   f2.add(8)
```

On the facing page (page 5), you are given an incomplete environment diagram representing the state of this program after line 12 has been executed, which you should complete. You do not need to include frames that no longer exist after line 12 has been run, but you are welcome to include them if you wish.


FRAMES
OBJECTS
GF
Foo
class
__init__
add
func
self
self.L = []
func
x, elt
self.L.append(x)

2 HTML Elements

HTML is a language that is used on the web, to tell your web browser how to display certain kinds of content. In its simplest form, it consists of *tags* (keywords that tell your browser how to display) surrounding content to which that tag should apply. For example, the following is valid HTML indicating that the text "hello!" should be displayed in bold (because of the `b` in the tag):

```
<b>hello!</b>
```

In general, an HTML element has the form above, with an opening tag that looks like `<type>`, followed by some content (which may contain other HTML elements), followed by a closing tag that looks like `</type>`. While writing HTML by hand is possible, it is often useful to use a framework that will do some of the work for you. What follows is a small framework designed to help with creating properly formatted HTML:

```
functions = {}

TAGS = [('b', 'bold'), ('u', 'underline'), ('i', 'italics')]

for (tag, word) in TAGS:
    functions[word] = lambda text: "<" + tag + ">" + text + "</" + tag + ">"
```

The intention is that, for example, calling `functions["italics"]` with a string as input should produce a new string containing that given input, surrounded with `<i>` and `</i>`.

The code above runs to completion (i.e., it does not produce an exception when run). However, note that there is **at least one issue** with the code above, such that the functions do not always produce the correct output.

Answer the questions on the facing page (page 7) about this code as written.

For each expression below, indicate both the type and the value that results from evaluating that expression in the box below it. If an expression would lead to an exception, briefly describe the type and nature of the exception. If it would lead to an infinite loop, write "infinite loop" instead.

```
>>> functions['bold']('hello')
```

```
>>> functions['underline'](42)
```

```
>>> functions['underline'](functions['bold']('hello'))
```

```
>>> (functions['underline'] + functions['bold'])('hello')
```

```
>>> functions['underline' + 'bold']('hello')
```

```
>>> functions['underline']('hello') + functions['bold']('hello')
```

3 Image Representations

The last page of this handout (which you may remove) contains code to represent greyscale images using a class called `Image`, which uses a similar internal representation to what we saw in lab 1.

Note that the initializer for this class takes as input three values: the height of the image, the width of the image, and a list of pixel values (in row-major order, as in lab 1). The implementation given on the last page of this handout stores the list of pixel values directly and computes everything from that list. However, this is not the only internal representation we could have used.

In this problem, we would like to implement an alternative representation as a class `ArrayImage`, which supports all of the same operations that `Image` does but uses a different internal representation to do so (a list-of-lists of pixel values, with each inner list containing the pixel values for a single row).

For example, the following pieces of code should result in objects that have different internal representations but on which all operations produce equivalent results:

- `Image(3, 2, [0, 50, 65, 100, 80, 255])`
- `ArrayImage(3, 2, [0, 50, 65, 100, 80, 255])`

On the facing page (page 9), fill in the definition of the `ArrayImage` class, which should use a list-of-lists as its internal representation, as described above (it should not store the row-major order representation).

Note that `ArrayImage` is a subclass of `Image`, so you do not necessarily need to reimplement all of the methods from `Image`. For full credit, you should only implement those pieces that are necessary for the class to function.

```
class ArrayImage(Image):
```

4 Linked Lists

As we saw in week 7's lecture, we can represent linked lists using classes. But, as we saw in lab 9, it is also possible to represent them without creating any classes. This problem explores one particular representation of linked lists that does not use classes. Instead, we use nested lists of the form `[element, rest]`, where `element` is the element **at the end of the list**, and `rest` is a linked list containing the remaining elements. We'll use `[None]` to represent an empty linked list. According to this representation:

- `[None]` represents an empty linked list;
- `[1, [None]]` represents a linked list containing only `1`; and
- `[9, [8, [7, [None]]]]` represents a linked list containing `7`, `8`, and `9`, **in that order**.

Part a. Length

Below is an implementation for a function that takes a list (of the form described above) as input and returns the length of the linked list it represents:

```
def length(L):
    if L == [None]:
        return 0
    return 1 + length(L[1])
```

One possible concern with this implementation relates to its efficiency, given the time and memory that creating new frames for each recursive call will take. However, it is also possible to write this function iteratively. In the box below, fill in an implementation of `length` that does not make any recursive calls.

```
def length(L):
```

Part b. Generator

We can also create a generator, which allows us to loop over the elements in one of our linked lists. Below is one implementation of such a generator. However, it does not quite work. It does not raise any exceptions when given a well-formed linked list as input, but it produces incorrect output for almost every nonempty linked list.

```
def gen(L):
    if L == [None]:
        return
    yield L[0]
    yield from L[1]
```

However, a few small changes (affecting no more than two lines in total) will fix the issue such that the resulting code will produce the correct output for all linked lists. What are the issues with the code, and what small changes can you make to fix them?

Part c. Insertion

We would also like to implement a function to insert elements into an existing linked list. This function should take three inputs: a linked list `L` (represented as in the previous parts), an `index` (*before* which the given element should be inserted), and the element to be inserted (`elt`). The function should always return `None`, but it should mutate the given linked list such that the new element `elt` now appears at the given `index`, shifting all later elements down.

Here is a small example showing the desired behavior of the function:

```
>>> test_list = [9, [8, [7, [None]]]]
>>> insert(test_list, 0, 99)
>>> test_list
[9, [8, [7, [99, [None]]]]]
>>> insert(test_list, 2, 98)
>>> test_list
[9, [8, [98, [7, [99, [None]]]]]]
>>> insert(test_list, 4, 97)
>>> test_list
[9, [97, [8, [98, [7, [99, [None]]]]]]]
```

Fill in the implementation of `insert` in the box on the facing page (page 13). Feel free to implement your solution recursively or iteratively (do not worry about the efficiency concerns from part a), and feel free to use any functions from the previous sections.

```
def insert(L, index, elt):
```

5 Potluck

You are organizing a potluck for your friends, and you want to make sure that everyone who comes to your party has something that they can eat! Given a dictionary containing food preferences (mapping a person's name to a list of food items that person is willing to eat) and the contents of your pantry (a list containing names of food items, where each food item may appear more than once), the function below is intended help with this task by returning a dictionary mapping each person's name to a single food item from their choices, subject to the constraint that the total number of people receiving each food item cannot exceed the number of that food item specified in the input list. If no such mapping is possible, the function should return `None`.

Some examples of the intended usage of this function are given on the second-to-last page of this handout, which you may remove.

```
01 |   def feed(people, foods):
02 |       if not people:
03 |           return {}
04 |
05 |       for person in people:
06 |           newpeople = {k: v for k,v in people.items() if k != person}
07 |           for ix, food in enumerate(people[person]):
08 |               if food in foods:
09 |                   newfoods = foods[:]
10 |                   newfoods.remove(food)  # remove the first occurrence of food from the list
11 |                   subresult = feed(newpeople, newfoods)
12 |                   if subresult:
13 |                       subresult[person] = food
14 |                       return subresult
15 |                   else:
16 |                       return None
17 |               return None
```

This implementation does not produce the correct output for all inputs, but it can be made to work with a small number of changes. In the boxes on the facing page (page 15), indicate up to five lines of code that you want to change (by indicating the line number and what you want to replace that line with), in the following way:

- If you want to replace the line with specific code, write that code in the box.
- If you want to change a line by deleting it (rather than by replacing it with other code), write "DELETE" in the box.
- If you want to change a line by indenting it further (but otherwise not changing the code), write "INDENT" in the box, along with the number of levels of indentation you would like to add.
- If you want to change a line by removing some amount of indentation (but otherwise not changing the code), write "DEDENT" in the box, along with the number of levels of indentation you would like to remove.

If you want to change fewer than 5 lines, please leave the remaining boxes blank.

Change line number to:

Change line number to:

Change line number to:

Change line number to:

Change line number to:

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Examples For Question 5

Here are some examples of usage of the `feed` function from Question 5 ("Potluck"). Note that these describe the **intended** behavior of the function, not necessarily what the code given in the question would produce.

```
>>> feed({'adrian': ['pickles'], 'bailey': ['apples']}, ['pickles', 'apples'])
{'adrian': 'pickles', 'bailey': 'apples'}

>>> feed({'adrian': ['pickles'], 'bailey': ['pickles']}, ['pickles', 'apples'])
None

>>> feed({'adrian': ['pickles'], 'bailey': ['pickles']}, ['pickles', 'apples', 'pickles'])
{'adrian': 'pickles', 'bailey': 'pickles'}

>>> people = {'adrian': ['pickles', 'apples'], 'bailey': ['chips', 'onions'],
...           'cameron': ['pie', 'broccoli'],  'dana': ['pickles'],
...           'emerson': ['onions'], 'frankie': ['pie']}
>>> foods = ['pickles', 'apples', 'chips', 'onions', 'pie', 'broccoli']
>>> feed(people, foods)
{'adrian': 'apples', 'bailey': 'chips', 'cameron': 'broccoli', 'dana': 'pickles',
 'emerson': 'onions', 'frankie': 'pie'}

>>> people = {'adrian': ['cake', 'cheese', 'pie', 'sandwiches'],
...           'bailey': ['cake', 'cheese', 'pie'],
...           'cameron': ['cake', 'cheese'],
...           'dana': ['cake', 'cheese'],
...           'emerson': ['cake', 'cheese']}
>>> foods = ['cake', 'cheese', 'pie', 'sandwiches', 'cake']
>>> res = feed(people, foods)
>>> res['adrian'], res['bailey']
('sandwiches', 'pie')

>>> sorted((res['cameron'], res['dana'], res['emerson']))
['cake', 'cake', 'cheese']
```

Worksheet (intentionally blank)

Code for `Image` Class

```
from PIL import Image as PILImage  # this import is only used in the save method below

class Image:
    def __init__(self, height, width, pixels):
        self.width = width
        self.height = height
        self.pixels = pixels

    def set_pixel(self, r, c, value):
        pixel_index = r * self.width + c
        self.pixels[pixel_index] = value

    def get_pixel(self, r, c):
        pixel_index = r * self.width + c
        return self.pixels[pixel_index]

    def empty_like(self):
        # returns an instance of whatever class self was an instance of, with
        # the same size but with all pixels set to 0
        return self.__class__(self.height, self.width, [0 for _ in self.flat_representation()])

    def flat_representation(self):
        return self.pixels[:]

    def copy(self):
        return self.__class__(self.height, self.width, self.flat_representation())

    def apply_per_pixel(self, func):
        for r in range(self.height):
            for c in range(self.width):
                self.set_pixel(r, c, func(r, c, self.get_pixel(r, c)))

    def transpose(self):
        out = self.__class__(self.width, self.height, [0 for _ in self.flat_representation()])
        out.apply_per_pixel(lambda r, c, val: self.get_pixel(c, r))
        return out

    def save(self, filename):
        out = PILImage.new(mode='L', size=(self.width, self.height))
        out.putdata(self.flat_representation())
        out.save(filename)
        out.close()
```

Worksheet (intentionally blank)