Object-Oriented Programming
50:198:113 (Spring 2016)

Homework:	**3**	**Professor:**	**Suneeta Ramaswami**
Due Date:	**3/22/16**	**E-mail:**	`suneeta.ramaswami@rutgers.edu`
Office:	**321 BSB**	**URL:**	`http://crab.rutgers.edu/~rsuneeta`
		Phone:	**(856)-225-6439**

Homework Assignment 3

The assignment is due by 11:59PM of the due date. The point value is indicated in square braces next to each problem. Each solution must be the student's own work. Assistance should only be sought or accepted from the course instructor. Any violation of this rule will be dealt with harshly.

This assignment contains one problem on recursion and two problems on class implementations. As usual, you are graded not only on the correctness of the code, but also on clarity and readability. Hence, I will deduct points for poor indentation, poor choice of object names, and lack of documentation. For documentation, all functions, classes, and methods should have appropriate docstring documentation. For the rest of your code, use a common sense approach. While I do not expect every line of code to be explained, all code blocks that carry out a significant task should be documented *briefly* in clear English.

Please read the submission guidelines at the end of this document before you start your work.

Important note: When writing each of the following programs, it is important that you name the functions exactly as described because I will assume you are doing so when testing your programs. If your program produces errors because the functions do not satisfy the stated prototype, points will be deducted.

Problem 1 [24 points] Recursive functions. In this problem, you are asked to write three recursive functions. In each of these functions, you **may not** use any built-in functions other than `len`, and the operators `[]`, `[:]`, and `+` (string or list concatenation). You also may not use any loops (the built-in function `in` counts as a loop).

1. *(8 points)* Write a recursive function called `replace_element` with three parameters: `L` (a list), `oldel`, and `newel`. The function returns a list in which every occurrence of `oldel` in `L` has been replaced with `newel`. For example, `replace_element([5, 4, 3, 5, 1], 5, 100)` should return the list `[100, 4, 3, 100, 1]`. Keep in mind that all restrictions stated above apply for the recursive implementation.
2. *(8 points)* Write a recursive function called `inverse_pair` with a single parameter `L`, which is a list of integers. The function returns `True` if `L` contains a pair of integers whose sum is zero and `False` otherwise. The base case occurs when the list has exactly two integers (since it doesn't make sense to talk about a "pair" of integers for lists with fewer than two elements). For example, `inverse_pair` should return `False` for the list `[12, 8, 10, -5]` and `True` for the list `[12, 5, 10, -5, -9]`. Your function should consist **only** of the base case and the recursive calls. As above, all usual restrictions apply for the recursive implementation.

3. *(8 points)* Write a recursive function called `occurrences` with two parameters: a string `astr` and another (nonempty) string `substr`. The function returns the number of times the substring `substr` appears in the string `astr`. For example, `occurrences("how now brown cow", "ow")` should return 4 and `occurrences("house mouse louse", "ow")` should return 0. As above, all usual restrictions apply for the recursive implementation.

Problem 2 [26 points] Straight Lines. In this problem, you are asked to implement a class for straight lines called `StraightLine`. As you know, a straight line in two dimensions is represented in standard form by the linear equation $ax + by = c$. In this representation, a is called the x coefficient, b is called the y coefficient, and c is called the constant coefficient. Thus the main instance attributes for straight line objects will be the three coefficients a, b, and c.

Implement the following methods for the `StraightLine` class. Recall once again that `self` is always the first parameter for a method of the class. **Your implementation should appear in a module called `problem2.py`.** A test module called `test2.py` is also provided, which imports your `problem2.py` module. When you are ready, test your implementation of the `StraightLine` class by typing `python3 test2.py`.

1. `__init__`: The constructor should create three instance attributes, which are the coefficients a, b, and c of the standard form equation of the line. The parameters to the constructor are initial values for these coefficients. Provide default values for all three attributes.
2. `__str__`: This method returns a printable version of a straight line, which is a "nice" string representation of the linear equation. This means that only non-zero coefficients of x and y should be printed, and if a coefficient is negative, the `-` (minus sign) must be embedded in the string. For example, the straight line $2x - 5y = 4$ should be represented by the following string (note that $a = 2, b = -5$, and $c = 4$ for this line):

   ```
   2x - 5y = 4
   ```

   and **not** as

   ```
   2x + -5y = 4
   ```

   Similarly, the straight line $5x = -6$ should be represented as

   ```
   5x = -6
   ```

   and **not** as

   ```
   5x + 0y = -6
   ```

3. `__repr__`: This method returns a meaningful string representation of the straight line. For example, this could be the same string returned by the `__str__` method.
4. `slope`: This method returns the slope of the line. If the line is vertical, the slope is undefined; the method should return `None` in this case.
5. `yintercept`: This method returns the y intercept of the line. If the line is vertical, then return `None` (since there is no y intercept for vertical lines).
6. `xintercept`: This method returns the x intercept of the line. If the line is horizontal, then return `None` (since there is no x intercept for horizontal lines).

7. `parallel`: This method has a parameter L which is another `StraightLine` object. It returns `True` if the line is parallel to L and `False` otherwise.
8. `perpendicular`: This method has a parameter L which is another `StraightLine` object. It returns `True` if the line is perpendicular to L and `False` otherwise.
9. `intersection`: This method has a parameter L which is another `StraightLine` object. It returns the point of intersection between the line and `L`. Return the point as a 2-tuple. If `L` is parallel to the line, then your function should return `None` (since two parallel lines do not intersect).
10. `__eq__`: This method overloads the equality operator `==`. We say that two `StraightLine` objects are equal if and only if they have the same slope and y-intercept. Make sure that your code handles vertical lines correctly.
11. `__ne__`: This method overloads the inequality operator `!=`.

Problem 3 [50 points] Container objects. In this problem, you are asked implement a class called `Container` in a module called `problem3a.py`. A `Container` object is an object that stores an unordered collection of items. An item may occur several times in a container. Note than a container may contain an arbitrary collection of objects (for example, a container may contain integers, strings, lists, or other objects). You are asked to implement the following methods for the `Container` class (recall that all methods have `self` as the first parameter):

1. `__init__`: The constructor creates an empty container. We store the contents of the container in a list. Hence, the constructor will simply create an instance attribute called `self.contents`, say, and initialize it as an empty list.
2. `insert`: This method has a single parameter `item` which is inserted into the container.
3. `erase_one`: This method has a single parameter `item`. One occurrence of `item` is deleted from the container. Recall that since the items in a container are unordered, it doesn't matter which occurrence is deleted. The container remains unchanged if `item` does not occur in the container.
4. `erase_all`: This method has a single parameter `item`. All occurrences of `item` are deleted from the container.
5. `count`: This method has a single parameter `item`. The number of occurrences of `item` in the container is returned.
6. `items`: This method has no parameters. A list of the *distinct* items in the container is returned. For example, if a container `C` has the items `1, "abc", 1, 2, 3, [4,5,6], 2, "abc", [4], [4,5,6]`, then `C.items()` should return `1, "abc", 2, 3, [4,5,6],[4]`. The order of items in the returned list does not matter.
7. `__str__`: Returns a string representation of the container. Recall that the return value must be a string. Make sure that multiple occurrences of the same item appear consecutively in the returned string. Other than that, the order of the items does not matter. *Note that you cannot sort the list, since it is an arbitrary collection of items which may not be comparable.*
8. `__repr__`: Also returns a printable representation of the container.
9. `__add__`: This method overloads the `+` operator and returns the "sum" of two containers. The sum of two containers is a new container that contains all the items of the two operand containers. Hence, if `A` is a container with items `1, 2, "abc", 2, [4]` and `B`

is a container with items `3, 2, 4, "abcd", [4], "xy"`, then `A + B` is the container with items `1, 2, "abc", 2, [4], 3, 2, 4, "abcd", [4], "xy"`. *Keep in mind that the actual parameters should not be modified by this method.*

10. `__sub__`: This method overloads the `-` operator and returns the "difference" of two containers. The difference of two containers `A` and `B` is a new container that contains only those items in `A` that do not occur in `B`. For example, if `A` and `B` are as shown in the example above, then `A - B` is the container `1, "abc"`, and `B - A` is the container `3, 4, "abcd", "xy"`. Again, the actual parameters should not be modified by this method.
11. `__mul__`: This method overloads the `*` operator and returns the "intersection" of two containers. The intersection of two containers `A` and `B` is a new container that contains items that are common to `A` and `B`. Note that the count of an item in the intersection is the minimum of the counts of that item in each of `A` and `B`. For example, if `A` and `B` are as shown in the example above, then `A * B` is the container with items `2, [4]`. Again, the actual parameters should not be modified by this method.

Containers as parameters to functions. After completing the above class implementation, you are asked to implement some functions that have containers as parameters and possibly as return values. Implement these functions in a module called `problem3b.py`. *On the first line of this module, import the* `Container` *class from the* `problem3a` *module.*

The following functions manipulate container objects. The `Container` class methods provide the interface required to manipulate objects of that class (the idea of *encapsulation* discussed in class), and you must use container methods to implement these functions. **Do not** manipulate container instance attributes directly. You will not receive any credit if you do.

- Implement a function called `symmetric_difference` with two parameters, a container `c1` and a container `c2`. The function returns a container that contains items that belong to `c1` or to `c2`, but not to both. For example, if `A` and `B` are the containers used in the examples in Problem 2, `symmetric_difference(A, B)` should return the container with items `1, "abc", 3, 4, "abcd", "xy"`.
- Implement a function called `subcontainer` with two parameters, a container `c1` and a container `c2`. The function returns `True` if `c1` is a subcontainer of `c2` and `False` otherwise. A container `A` is said to be a subcontainer of container `B` iff every item in `A` is also in `B`. Note that this definition implies that the count of an item in `A` has to be at most its count in `B`.
- Implement a function called `remove_repeats` with a single parameter, a container `C`. The function removes all but one copy of each item in the container. That is, repetitions of an item are removed from the container. Note that container `C` will be modified by this function.
- Implement a function called `similar` with two parameters, a container `A` and a container `B`. The function returns `True` if the two containers are "similar" and `False` otherwise. Two containers are said to be similar if they contain exactly the same items (but the counts of the items may be different in the two containers). Hence, a container with the items `1, 2, 2, 3, 4, 4, 4` is said to be similar to a container with the items `1, 2, 3, 3, 4`.

Submission Guidelines

Implement the first problem in a module called `problem1.py` and the second one in a module called `problem2.py`. The third problem requires you to create two modules: `problem3a.py` and `problem3b.py`. *Your name and RUID should appear as a comment at the very top of each file.*

Test each of your programs thoroughly before submitting your homework. When you are ready to submit, upload your files on Sakai as follows:

1. Use your web browser to go to the website `sakai.rutgers.edu`.
2. Log in by using your Rutgers login id and password, and click on the `OBJECT-ORIENTED PROG S16` tab.
3. Click on the 'Assignments' link on the left and go to 'Homework Assignment #3' to find the homework file (`hw3.pdf`), the test file `test2.py` for problem #2, and the test file `test3.py` for problem #3.
4. Use this same link to upload your homework files (`problem1.py`, `problem2.py`, `problem3a.py`, and `problem3b.py`) when you are ready to submit.

You must submit your assignment at or before 11:59PM on March 22, 2016.