Object-Oriented Programming
50:198:113 (Fall 2018)

Homework Assignment 4

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 requires you to go further with classes. In Problem 1, you are asked to implement a new class called 

Trip

(which will have a Date object as an instance attribute). In Problem 2, you are asked to implement a class called 

TripSchedule

(this is a collection of Trip objects). In Problem 3, you are asked to design a new class called ChangeJar.

As usual, you are graded not only on the correctness of the code, but also on clarity and readability. I will deduct points for not following the guidelines for your class design, poor indentation, poor choice of object names, and lack of documentation. For documentation, 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.

Problem 1 [35 points] Trip. In a company, it is frequently necessary to keep track of the travel schedule for an employee (henceforth referred to as a person) whose job requires frequent travel. In this problem, you are asked to implement a class called Trip, described below, that will help the company do this. Create a module called trip.py to contain this class. (In the next problem, you will use the Trip class to implement a container class to hold multiple trips for an employee.)

Important: You will need the Date class to implement this class. I am providing you with my implementation of the Date class in the module date.py. Please use this module for your implementation of the Trip class (do not use your own Date class) as it will allow for standardized testing of your work. Make sure that you include the line from date import Date at the top of your trip.py module.

Details about the Trip class implementation are as follows. An instance of the Trip class allows us to store and retrieve information about a specific trip. The attributes of a trip are as follows (they should all be private):

- destination, the destination city of the trip (this is a string). To keep things simple, we’ll assume the person travels to only one city at a time,
- depdate, the on which the person departs on the trip (this is a Date object),
- duration, the duration of the trip; that is, the number of nights the person is away from the home town (this is an integer ≥ 1).

We include various methods that manipulate attributes of this class, as described below.
1. A constructor to initialize the three trip instance attributes described above. Keep in mind that `depdate` is a `Date` instance.

2. A method `setDestination` to set the trip destination to a given value (a string).

3. A method `setDeparture` to set the trip departure date to a given value (a `Date`).

4. A method `setDuration` to set the trip duration to a given value (an integer).

5. A method `destination` that returns the destination of the trip.

6. A method `departure` that returns the departure date of the trip.

7. A method called `duration` that returns the duration of the trip.

8. A method `arrival` that returns the arrival date for the trip (the date on which the person arrives back to the home town). Note that the return value is a `Date`. Use `Date` methods to implement this function. Do not repeat code needlessly.

9. A method called `overlaps` with two parameters `self` and `other`, where `other` is also a `Trip`. The method returns `True` if the trips `self` and `other` overlap. Note that two trips are considered to overlap if the dates of travel (including departure and arrival dates) of one overlap with the dates of travel of the other. Hint: The comparison operators for `Date` objects make this function quite easy to implement.

10. A method called `containsweekend` that returns `True` if the trip contains at least one day of a weekend (Saturday or Sunday) and `False` otherwise. For example, if a trip `T` to London was 2 days long and started on March 26, 2015, then `T.containsweekend()` will return `True`. However, if `T` started on March 24, 2015, then `T.containsweekend()` will return `False`.

11. A method `__str__` to print the trip details in a neatly formatted way. The trip details include the destination, the duration of the trip, the departure date (indicate the day of week as well), and the arrival date (again, indicate the day of week). Here is an example output:

   Destination: Paris
   Duration: 6 days
   Departure: Sunday, April 19, 2015
   Arrival: Saturday, April 25, 2015

Problem 2 [30 points] TripSchedule. You will implement this class in a module called `tripschedule.py`. This class requires you to use the `Trip` class from Problem 1. Make sure that you insert the following line at the top of your `tripschedule.py` module:

   from trip import Trip

An instance of the `TripSchedule` class stores a collection of trips that form the trip schedule for one person. Note that the trips must conform to the following consistency requirement: no two trips can conflict with one another. This means that the dates on which an employee is traveling on one trip cannot overlap with the dates on which s/he is traveling on another. We also do not allow a person to depart on a trip on the same day s/he arrives back from another.

As the description suggests, the `TripSchedule` class will be a “container” for other objects (in particular, for `Trip` objects). You are required to include the following methods for this class:
• A constructor that creates an empty trip schedule. Store the trips in the schedule in a list.

• A method called insert to add a new trip to the schedule if it does not conflict with existing ones. If there is a conflict, print an appropriate message (no need to raise an exception). Hint: The Trip method overlaps will be useful here.

• A method called delete to delete a trip from the schedule.

• A method called __len__ (this overloads the built-in len() function) to return the length of the trip schedule (i.e., the total number of trips in the schedule).

• A method called __getitem__ to overload the index operator. An index value of j returns the j-th trip in the schedule (use the index 0 for the first trip, 1 for the second trip, and so on...)

• A method called search to search the schedule by a keyword that can be either a destination or a month. (Therefore, this method has two parameters: self and keyword.) If the keyword is an integer in the range 1 to 12 (inclusive), then all trips in the schedule that start in that month should be printed out. Otherwise, the keyword is assumed to be a destination and all trips in the schedule with that destination should be printed out. Use the print function for Trips for proper display. Extra credit (5 points): Print the the trips in sorted order of departure date.

• A method called available with three parameters self, month, and year to search the schedule for all available dates in month (an integer between 1 and 12) of year. Available dates are dates on which there is no travel scheduled. The function returns a list of all available dates in month of year.

• A method __str__ to return a string representation of the trip schedule. Hint: Use the str function for Trip objects for a straightforward implementation of this method.

Problem 3 [35 points] ChangeJar. In this problem, you are asked to create a class for change jars. A change jar contains an arbitrary collection of coins i.e., quarters, dimes, nickels, and pennies. There are no dollar bills of any denomination in a change jar. We may use change jars to get exact change for some specified amount, or we may add more coins into it. Create a module called changejar.py to implement these and other methods for change jars in a class called ChangeJar. Further details are provided below:

1. __init__: The constructor has a single parameter, which is a dictionary of key:value pairs in which the keys must be 25, 10, 5, or 1 (for quarters, dimes, nickels, and pennies). All other keys are invalid (you may raise an exception in this case). The value associated with each key is the number of coins of that denomination in the change jar. Note that not every key need appear in the parameter. If a key does not appear, it means that there are no coins of that denomination in the jar. Set the default value of the parameter to be the empty dictionary, which will allow us to create change jars with no quarters, dimes, nickels, or pennies.

   The constructor should create a dictionary as an instance attribute which has exactly four keys: 25, 10, 5, and 1. The value associated with each key should be the number of coins of that type in the jar.

   For example, the statement J = ChangeJar({25:8, 5:10, 1:45}) creates a change jar that has 8 quarters, 0 dimes, 10 nickels, and 45 pennies. The statement J = ChangeJar() creates a change jar with 0 quarters, 0 dimes, 0 nickels, and 0 pennies.
2. **get_change**: This method has a single parameter, `dollar_amt`, which is a value corresponding to a dollar amount. It should **return** a `ChangeJar` object that contains the number of quarters, dimes, nickels, and pennies required to create **exact** change corresponding to that dollar amount. The corresponding numbers of coins should be deducted from the activating change jar.

There is more than one way to make exact change for a specified value. Here, you are used to use the fewest number of coins possible to make the change. Note that you must return exact change. This means that if the coins in the jar cannot form exact change for `dollar_amt`, the method should return an empty change jar. Keep in mind that this may happen even if there is enough total money in the change jar. For example, if a change jar has 4 quarters, 3 dimes, and 4 pennies, we cannot get exact change for $1.25 from the jar, even though it has $1.34 in it.

**Tip:** Convert `dollar_amt` to cents before you find the change. That is, work with integer values, rather than real values.

**Note:** This is the most challenging method to implement. All the others are quite straightforward, so it is suggested you implement all the others before this one.

3. **getitem**: This method overloads the index operator. When the index value is 25, 10, 5, or 1, the method returns the number of quarters, dimes, nickels, or pennies, respectively, in the jar. All other index values smaller than 25 should return 0, and an index value greater than 25 should raise the `IndexError` exception.

4. **insert**: This method is used to add more coins of a particular value into the change jar. The method has two parameters: `coin_value` (which has value 25, 10, 5, or 1) and `num_coins` (the number of coins of that value being inserted into the jar). Note that you are adding to the existing coins in the jar. For example, if `J` is a `ChangeJar` instance, `J.insert(10, 12)` will insert 12 more dimes into `J`.

5. **total_value**: This method returns the total dollar value of the change jar as a real number.

6. **str**: This method returns a string representation of the change jar. Recall that this method must return a string. It should contain succinct information about the number of quarters, dimes, nickels, and pennies in the change jar.

7. **repr**: This method returns a printable representation (also a string) of the change jar.

8. **add**: This method overloads the `+` operator. It returns a change jar that contains all the coins from the two change jars that are the operands. Keep in mind that the operands themselves should not get modified by this method.

9. **eq**: This method overloads the `==` operator. It returns `True` if the two change jars have exactly the same numbers of coins of each type, and `False` otherwise.

10. **ne**: This method overloads the `!=` operator. It returns `True` if the two change jars are not equal (as defined above) and `False` otherwise.

**Submission Guidelines**

Please name the module for each problem as specified in the problem description above. In particular, create a module called `trip.py` for Problem 1, a module called `tripschedule.py` for Problem
2, and a module called changejar.py for Problem 3. Also, please make sure that your name and RUID appear as a comment at the very top of the file.

Submit your homework files via 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 F18 tab.

3. Click on the 'Assignments' link on the left and go to 'Programming Assignment #4' to find the homework file (hw4.pdf) and the module date.py (this will be made available after November 3, to allow possible late submissions of Homework 3). I will also provide test files for your class implementations in a few days.

4. Use this same link to upload your three homework files (trip.py, tripschedule.py, and changejar.py) when you are ready to submit.

You must submit your assignment at or before 11:55PM on November 25, 2018.