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

Homework:	**4**	**Professor:**	**Suneeta Ramaswami**
Due Date:	**4/13/16**	**E-mail:**	`suneeta.ramaswami@rutgers.edu`
Office:	**321 BSB**	**URL:**	`http://crab.rutgers.edu/~rsuneeta`
		Phone:	**(856)-225-6439**

Homework Assignment 4

The assignment is due by 11:55PM 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 implement your own classes. 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 [**40 points**] **Calendar dates.** In this problem, you are asked to implement a class called `Date` for calendar dates occurring on or after January 1, 1800. (Python has its own `datetime` module, but, needless to say, you **should not** use that when implementing this class.) Instances of the `Date` class have a month, day, and year value representing valid calendar dates on or after January 1, 1800. Create a module called `problem1.py` to contain your `Date` class implementation. Details about the class and its methods are provided below:

1. Assign a class attribute called `min_year` the value 1800. This represents the smallest year value allowed for instances of the `Date` class. The benefit of using a name to refer to the minimum year is that if we change our mind about the smallest allowable year value, we need to change it in only one place. Use `min_year` everywhere to refer to this value, rather than hard-coding 1800 in your code. In addition, assign another class attribute called `dow_jan1` the value `''Wednesday''`. This represents the day of week on January 1 of the year `min_year`.
2. `__init__`: The constructor sets the values of the month, day, and year attributes of the date. You are required to make all instance attributes private. The default values for these should be 1, 1, and `min_year`, respectively. **The constructor must check for the validity of the date**, and if the date is invalid, should raise an exception. This means that the month should lie between 1 and 12 (inclusive), the day should be valid *for that month*, and the year should be greater than or equal to `min_year`. For example, the dates 2/30/2008, 2/29/2009, and 9/31/2010 are all invalid. Make sure that you take leap years into account when determining the validity of the date. The method `is_leap` will come in handy here.
3. `month`: Returns the month of the date.

4. `day`: Returns the day of the date.
5. `year`: Returns the year of the date.
6. `year_is_leap`: Returns `True` if the year of the date is a leap year and `False` otherwise. A year is said to be a leap year if it is a multiple of 4. However, if it is also a multiple of 100, then it is a leap year *only* if it is a multiple of 400. So, for example, 1948 was a leap year, as will be 2124. However, 1800 and 1900 were not leap years, but 2000 was.
7. `daycount`: Returns the total number of days from January 1, 1800 to the date. For example, if `d` is the date 2/14/1801, then `d.daycount()` should return 410 (there are 410 days from January 1, 1800 to February 14, 1801). Make sure that you take leap years into account.
8. `day_of_week`: Returns the day of week of the date. Hence, the return value is one of `"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", or "Sunday"`. Please make sure that the day of week returned by your method is exactly one of the seven strings shown here; this will make it easier for me to test your method. Important fact: January 1, 1800 was a Wednesday. You will find the `daycount` method useful here.
9. `nextday`: Returns the date of the following day. For example, if `d` is the date 3/31/1801, `d.nextday()` should return the date 4/1/1801.
10. `prevday`: Returns the date of the previous day. Note that January 1, 1800 does not have a previous day. This error should be caught by raising an `Exception`.
11. `__add__`: This method overloads the `+` operator. It has two parameters: `self` and an integer `n`. It returns the date that occurs `n` days `after` the date `self`. For example, if `d` is the date 4/25/2015, then after the expression `newd = d + 9`, `newd` is the date 5/4/2015. You will find the method `nextday` useful here.
12. `__sub__`: This method overloads the `-` operator. It has two parameters: `self` and an integer `n`. It returns the date that occurs `n` days *before* the date `self`. For example, if `d` is the date 4/25/2015, then after the expression `newd = d - 25`, `newd` is the date 3/31/2015. You will find the method `prevday` useful here.
13. `__lt__`: This method overloads the `<` operator. It has two parameters: `self` and `other`, which is a date. It returns True if `self` comes *before* `other`.
14. `__eq__`, `__le__`, `__gt__`, `__ge__`, and `__ne__`: Overload all remaining relational operators as well, using the obvious interpretation of these operators for dates.
15. `__str__`: This method returns a printable (i.e., string) representation of the date. For example, if `d` is the date 4/25/2015, then `str(d)` should return the string `"April 25, 2015"`.
16. `__repr__`: This method also returns a string representation of the date.

Problem 2 [20 points] Functions that manipulate `Date` objects. In this problem, you are asked to implement three functions that manipulate `Date` objects. You must use Date class methods (as implemented in Problem 1 above) to implement all of the following functions. At the top of your `problem2.py` module, import the `Date` class using `from problem1 import Date`.

1. Implement a function called `weekend_dates` with two parameters, a month `m` ($1 \leq$ `m` ≤ 12) and a year `y`. The function should *print* all the weekend (Saturday and Sunday) dates that occur in month `m` of year `y`. For example, `weekend_dates(4, 2016)` should print

```
April 2, 2016 (Saturday)
April 3, 2016 (Sunday)
April 9, 2016 (Saturday)
April 10, 2016 (Sunday)
April 16, 2016 (Saturday)
April 17, 2016 (Sunday)
April 23, 2016 (Saturday)
April 24, 2016 (Sunday)
April 30, 2016 (Saturday)
```

2. Implement a function called `first_mondays` with a single parameter, namely a year `y`. The function should print the dates of the first Monday of every month in that year. For example, `first_mondays(2016)` should print

```
First Mondays of 2016:

January 4, 2016
February 1, 2016
March 7, 2016
April 4, 2016
May 2, 2016
June 6, 2016
July 4, 2016
August 1, 2016
September 5, 2016
October 3, 2016
November 7, 2016
December 5, 2016
```

3. There are many situations in which one needs a schedule of dates occuring at regular intervals between a start date and an end date. For example, a course instructor may want to give an online class quiz every 12 days starting on September 6, 2016 and ending on or before October 31, 2016. Implement a function called `interval_schedule` with three parameters: `start_date` (a `Date` object), `end_date` (a `Date` object), and `interval` (a positive integer). The function should **return** the list of dates that occur every `interval` days, starting on `start_date` and ending on or before `end_date`. *Note that the function is returning a list of* `Date` *objects.* For example, `interval_schedule(Date(9, 6, 2016), Date(10, 31, 2016), 12)` should return a list of `Date` objects. If you print the elements of this list, you should see:

```
September 6, 2016
September 18, 2016
September 30, 2016
October 12, 2016
October 24, 2016
```

Problem 3 [**40 points**] **Change Jar.** 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 `problem3.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.

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 `StopIteration` 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 `problem1.py` for Problem 1, a module called `problem2.py` for Problem 2, and a module called `problem3.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 S16` tab.
3. Click on the 'Assignments' link on the left and go to 'Homework Assignment #4' to find the homework file (`hw4.pdf`).
4. Use this same link to upload your homework files (`problem1.py`, `problem2.py`, and `problem3.py` when you are ready to submit.

You must submit your assignment at or before 11:55PM on April 13, 2016.