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

Homework: 3
Due Date: 10/31/2018
Office: 321 BSB
Professor: Suneeta Ramaswami
E-mail: rsuneeta@camden.rutgers.edu
URL: http://crab.rutgers.edu/~rsuneeta
Phone: (856)-225-6439

Homework Assignment 3

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 or TA only. Any violation of this rule will be dealt with harshly.

This assignment contains one problem on recursion, one problem on a Date class implementation, and one problem on functions that manipulate Date objects. 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 the triple-quote method for the entire module, as well as each function (Problems 1 and 3) or class (Problem 2); keep in mind that each method of the class should also be documented.

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 all functions, classes, and methods 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 [25 points | Recursive Functions. In this problem, you are asked to write three recursive functions. Implement all functions in a module called problem1.py.

1. (8 points) Write a recursive function called duplicates with a single parameter L, a list of items. The function returns True if L has any duplicate (i.e. repeating) items and False otherwise. The function must be implemented recursively. The base case occurs when the list is empty (in which case it returns False). Your function should consist only of the base case and recursive calls in an if/else statement. You are not allowed to use any built-in functions other than len for lists (please note that in is a built-in function, and you may not use it to implement your function). In addition, you are only allowed to use the index operator [] and the slice operator [:] for lists. There should not be any loops (for or while) in your implementation! Hint: Your function will need two recursive calls, not just one.

2. (8 points) Write a recursive function called count_pattern with a single parameter astr, which is a string. The function returns the number of times the substring “ou” appears in the string. The base case occurs when the string has length 2 or less. For example, count_pattern("house mouse louse") should return 3 and count_pattern("true blue") should return 0. Once again, your function should consist only of the base case
and recursive calls. Again, *no loops!* Do not use any builtin string functions other than \texttt{len}, or the string operators [ ] and [:] for indexing and slicing, respectively.

3. *(9 points)* Write a recursive function called \texttt{negative\_sum} with a single parameter \texttt{L}, which is a list of integers. The function returns \texttt{True} if \texttt{L} contains a pair of integers whose sum is negative and \texttt{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, \texttt{negative\_sum} should return \texttt{False} for the list \([12, 8, 10, -5]\) and \texttt{True} for the list \([12, 3, 8, 10, -5]\). Your function should consist only of the base case and the recursive calls. *No loops* should be used in the implementation. You are not allowed to use any builtin functions other than \texttt{len} (for the base case). In addition, you are only allowed to use the index operator [], the slice operator ::, and list concatenation.

**Problem 2 [45 points] Calendar dates.** In this problem, you are asked to implement a class called \texttt{Date} for calendar dates occurring on or after January 1, 1800. FYI, January 1, 1800 was a Wednesday. Instances of the \texttt{Date} class have a month, day, and year value representing valid calendar dates on or after January 1, 1800. Create a module called \texttt{problem2.py} to contain your \texttt{Date} class implementation. Details about the class and its methods are provided below. (Python has its own \texttt{datetime} module, but, needless to say, you \textbf{should not} use that when implementing this class.)

1. Assign a class attribute called \texttt{min\_year} the value 1800. This represents the smallest year value allowed for instances of the \texttt{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 \texttt{min\_year} everywhere to refer to this value, rather than hard-coding 1800 in your code. In addition, assign another class attribute called \texttt{dow\_jan1} the value `'Wednesday'`. This represents the day of week on January 1 of the year \texttt{min\_year}.

2. \texttt{\_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 \texttt{min\_year}, respectively. \textbf{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 \textit{for that month}, and the year should be greater than or equal to \texttt{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 \texttt{is\_leap} will come in handy here.

3. \texttt{month}: Returns the month of the date.
4. \texttt{day}: Returns the day of the date.
5. \texttt{year}: Returns the year of the date.
6. \texttt{year\_is\_leap}: Returns \texttt{True} if the year of the date is a leap year and \texttt{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 \textit{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. \texttt{daycount}: Returns the total number of days from January 1, 1800 to the date. For example, if \texttt{d} is the date 2/14/1801, then \texttt{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.

**Note:** I will shortly provide (on Sakai) a test file called *testdate.py* for you to test your implementation of the `Date` class.

**Problem 3 [30 points] Functions that manipulate Date objects.** In this problem, you are asked to implement three functions that manipulate `Date` objects in a module called `problem3.py`. You must use `Date` class methods (as implemented in Problem 2 above) to implement all of the following functions. At the top of your `problem3.py` module, import the `Date` class using `from problem2 import Date`.

1. Implement a function called `weekend_dates` with two parameters, a month `m` (1 ≤ 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 occurring 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

Submission Guidelines

Implement the first problem as `problem1.py`, the second one as `problem2.py`, and the third one as `problem3.py`. Your name and RUID should appear as a comment at the very top of each file. Points will be deducted if you do not follow the specified naming convention.

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 #3' to find the homework file (hw3.pdf). (*Note: In a day or two, I will also post a test file called testdate.py for Problem 2.*)

4. Use this same link to upload your three homework files (problem1.py, problem2.py, and problem3.py) when you are ready to submit.

You must submit your assignment at or before 11:59PM on October 31, 2018.