Object-Oriented Programming
50:198:113 (Fall 2020)

Homework:	**4**	**Professor:**	**Suneeta Ramaswami**
Due Date:	**11/8/20**	**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 11:59PM 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 be sought or accepted from the course instructor only. Any violation of this rule will be dealt with harshly.

The first problem of this assignment is a continuation of the Date class implementation from Homework 3, wherein you are asked to include the implementation of overloaded operators, arithmetic as well as comparison, for the `Date` class. The second problem asks you implement some functions that manipulate `Date` objects. The third problem requires you to implement a class for *polynomial equations*. 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. You are expected to provide docstring documentation for modules, classes and their methods, as well as functions.

Problem 1 [12 points] You are now asked to implement overloaded operators for the `Date` class from Homework 3. Please note that you are required to make all instance attributes private in this implementation.

If your implementation of the `Date` class in Homework 3 was completely correct, you may use your own implementation (but make sure you change your instance attributes to make them private). However, if it was not fully correct, please use my implementation provided under "Assignment #3 Solutions" on Canvas. This will allow me to grade only the newly inserted overloaded operators in this homework.

1. `__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.
2. `__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.
3. `__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`.
4. `__eq__`, `__le__`, `__gt__`, `__ge__`, and `__ne__`: Overload all remaining relational operators as well, using the obvious interpretation of these operators for dates.

You may once again test your implementation by using the test module `testdate.py` provided on Canvas.

Problem 2 [13 points] Functions that manipulate `Date` objects. In this problem, you are asked to implement three functions that manipulate `Date` objects in a module called `datefuns.py`. You must use Date class methods (as implemented in Problem 1 above) to implement all of the following functions. At the top of your `datefuns.py` module, import the `Date` class using `from date import Date`. **Important:** You may not manipulate instance attributes directly to implement these functions (which you cannot do anyway if you made all the instance attributes private). You must use `Date` methods.

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 [50 points] Polynomial objects. In this problem, you are asked to implement a class for *polynomials*. *Please read the description carefully*, as it will provide details about class methods and instance attributes.

A polynomial is an expression of the form

$$c_0 + c_1x + c_2x^2 + c_3x^3 + \ldots + c_nx^n$$

where $c_0, c_1, \ldots, c_n$ are real numbers. Each of c_ix^i is called a *term* of the polynomial, and c_i is called the *coefficient* of the term with *exponent* i. Note that c_0 is simply the coefficient of the term with exponent 0. The maximum exponent with a non-zero coefficient is called the *degree* of the polynomial. For example, $6x^{14} + 9x^{11} - 12x^3 + 42$ is a polynomial of degree 14. The polynomial $-12x^6 + 5x^5 - 20x^4 + 8x^2 - 12x + 9$ has degree 6. The following are standard operations on polynomials:

Scaling a polynomial: Given a polynomial $p(x) = c_nx^n + c_{n-1}x^{n-1} + \ldots + c_1x + c_0$ and a real value s, *scaling* $p(x)$ by s gives the polynomial $s \cdot p(x)$ obtained by scaling the coefficient of every term in $p(x)$ by the factor s. For example, if $p(x) = 6x^{14} + 9x^{11} - 12x^3 + 42$, then $2p(x) = 12x^{14} + 18x^{11} - 24x^3 + 84$.

Sum of two polynomials: The sum of two polynomials $p_1(x)$ and $p_2(x)$, denoted by $p_1(x) + p_2(x)$, is the polynomial obtained by adding the terms of $p_1(x)$ and $p_2(x)$. For example, if $p_1(x) = 6x^5 - 4x^3 + 10x^2 + 4$ and $p_2(x) = 3x^9 - 5x^7 - 3x^2 + 8x$, then $p_1(x) + p_2(x) = 3x^9 - 5x^7 + 6x^5 - 4x^3 + 7x^2 + 8x + 4$.

Difference of two polynomials: The difference of two polynomials $p_1(x)$ and $p_2(x)$ is the polynomial obtained by subtracting one from the other. Therefore, $p_1(x) - p_2(x)$ is the polynomial obtained by subtracting the terms of $p_2(x)$ from $p_1(x)$. For example, if $p_1(x) = 6x^5 - 4x^3 + 10x^2 + 4$ and $p_2(x) = 3x^9 - 5x^7 - 3x^2 + 8x$, then $p_1(x) - p_2(x) = -3x^9 + 5x^7 + 6x^5 - 4x^3 + 13x^2 - 8x + 4$.

Product of two polynomials: The product of two polynomials $p_1(x)$ and $p_2(x)$, denoted by $p_1(x) \cdot p_2(x)$, is the polynomial obtained by the pair-wise multiplication of terms in $p_1(x)$ and $p_2(x)$. For example, if $p_1(x) = x^4 + 4x^3 + 4x^2$ and $p_2(x) = 2x - 1$, then $p_1(x) \cdot p_2(x) = 2x^5 + 7x^4 + 4x^3 - 4x^2$. This is obtained as follows: $(x^4 + 4x^3 + 4x^2)(2x) + (x^4 + 4x^3 + 4x^2)(-1) = 2x^5 + 8x^4 + 8x^3 - x^4 - 4x^3 - 4x^2 = 2x^5 + 7x^4 + 4x^3 - 4x^2$.

In this program, you are asked to implement a `Polynomial` class. The instance attribute for an object of this class is a list that stores the coefficients of the polynomial. A polynomial of degree d is stored in a list of length $d + 1$. For each term c_ix^i, coefficient c_i is stored at index i in the list. Note that if a term of exponent i does not exist in the polynomial, its coefficient is 0 and hence the element at index i in that case will be 0. For example, the polynomial $-3x^9 + 5x^7 + 6x^5 - 4x^3 + 13x^2 + 4$ will be stored in a list of length 10 as follows: `[4, 0, 13, -4, 0, 6, 0, 5, 0, -3]`.

Implement the following methods for the `Polynomial` class. *Your implementation should appear in a module called* `poly.py`. A test module called `testpoly.py` is also provided, which imports your `poly.py` module. When you are ready, you may test your implementation of the `Polynomial` class by running the `testpoly.py` module at the command line.

1. `__init__`: The parameters to the constructor are a number of term pairs, where each term pair is a 2-tuple of the form (coefficient, exponent). Note that the number of such pairs is unspecified, since a polynomial may have any number of terms. Hence, the `def` statement for the constructor will be as follows:

   ```
   def __init__(self, *termpairs):
   ```

   For example, to create a `Polynomial` instance for the polynomial $6x^{14}+9x^{11}-12x^3+42$, we would do the following:

   ```
   P = Polynomial((6, 14), (9, 11), (-12, 3), (42, 0))
   ```

   Note that the term pairs can be specified in any order. Hence, `Polynomial((6, 14), (9, 11), (-12, 3), (42, 0))` and `Polynomial((42, 0), (6, 14), (-12, 3), (9, 11))` are the same polynomial. The constructor should create an instance attribute called `coeffs`, which is a list that stores coefficients of terms in the polynomial as described above. Keep in mind that

   - each term pair has the coefficient as the first element of the 2-tuple and the exponent as the second element,
   - the degree of the polynomial is the highest exponent in all the term pairs, and
   - the list in which you store the coefficients must have length equal to the degree of the polynomial plus 1.

2. `__str__`: This method returns a printable version of a polynomial, which is a "nice" string representation of the polynomial. This means that only terms with non-zero coefficients must appear in the string, and the terms must appear from largest exponent to smallest. Furthermore, if the coefficient of a term is negative, the `-` (minus sign) must be embedded in the string.

3. `__repr__`: This method returns a meaningful string representation of the polynomial. For example, this could be the same string returned by the `__str__` method.

4. `degree`: This method returns the degree of the polynomial.

5. `evaluate`: This method returns the result of evaluating the polynomial at a real value `x = X`. For example, if `P` is the polynomial $-12x^6+5x^5-20x^4+8x^2-12x+9$, then `P.evaluate(2)` should return -911 (which we obtain by plugging in 2 as the value of x). This method has two parameters: `self` and `X` (the real value at which the polynomial is to be evaluated).

6. `addterm`: This method adds a term with coefficient `cf` and exponent `exp` to the polynomial. If a term with this exponent already exists, the new term is simply added on to it. For example, if `P` is the polynomial $-4x^3+13x^2+4$, then `P.addterm(6, 5)` adds the term $6x^5$ to `P`, so that `P` is now $6x^5-4x^3+13x^2+4$. Subsequently, `P.addterm(-3, 2)` causes `P` to become $6x^5-4x^3+10x^2+4$. **Note** the order of the parameters: we specify the coefficient first and then the exponent. Keep in mind that you may need to increase or decrease the length of the list `coeffs` in this method. The length would need to increase if the term being added has exponent greater than the degree. The length

would need to decrease if the term being added results in the coefficient of the highest exponent term becoming 0.

7. `removeterm`: This method removes the term with exponent `exp` from the polynomial. If such a term does not exist, the polynomial remains unchanged. For example, if `P` is $6x^5 - 4x^3 + 10x^2 + 4$, `P.removeterm(3)` causes `P` to become $6x^5 + 10x^2 + 4$. Again, keep in mind that you may need to decrease the length of the list `coeffs` in this method (if the term being removed is the one with highest exponent).
8. `scale`: This method returns the result of scaling the polynomial by a factor `s`. **Note that a polynomial object is returned.** Moreover, the activating polynomial itself should remain unchanged. For example, if `P` is $6x^5 + 10x^2 + 4$, then `P.scale(2.5)` returns the polynomial $15x^5 + 25x^2 + 10$. `P` itself is unchanged.
9. `__add__`: This method overloads the `+` operator and returns the sum of two polynomials, which is also a polynomial (as explained above). Note that `__add__` has two parameters, `self` and `other`, where `other` is also a polynomial. Implement this function by *reusing* code already written, i.e., by calling methods already implemented. *Important:* Neither `self` nor `other` should be modified by this method.
10. `__sub__`: This method overloads the `-` operator and returns the difference of two polynomials, which is also a polynomial (as explained above). Comments on the `__add__` method apply here as well.
11. `__mul__`: This method overloads the `*` operator and returns the product of two polynomials, which is also a polynomial (as explained above). Again, neither `self` nor `other` should be modified by this method.
12. `__getitem__`: This method overloads the `[]` (indexing) operator. If `idx` is the parameter to the index operator, `__getitem__` must return the coefficient of the term with exponent `idx`. For example, if `P` is $6x^5 + 10x^2 + 4$, `P[5]` should return 6, `P[0]` should return 4, and `P[3]` should return 0.
13. `__setitem__`: This method implements the index assignment operation. If `idx` and `value` are the two parameters to this method, then `value` should become the coefficient of the term with exponent `idx`. If a term with this exponent already exists, its coefficient is changed to `value`. If a term with this exponent does not exist, it should be created.

In addition to the above `Polynomial` class methods, you must implement the following two functions (these are regular functions, not methods):

- A function called `read_polynomial` with a single parameter, a string `polyfilename`, which is the name of a file that contains the terms of a polynomial in the following format: The file has as many lines as there are terms, and each line of the file contains a pair of numbers, the first being the coefficient of the term and the second being the exponent. This function opens the file for reading, creates a `Polynomial` instance from the data in the file, and then **returns** this instance.

  Two files called `poly1.txt` and `poly2.txt` containing polynomial data have been created for you. They are available on Canvas along with the test files. You may use these to test your implementation. Try creating some of your own as well for further testing.
- A function called `arith_ops_polys` with two parameters `P` and `Q`, both of which are polynomial instances. This function simply prints (on separate lines) the degree of each polynomial, and the polynomials resulting from adding, subtracting, and multiplying the polynomials `P` and `Q`.

Submission Guidelines

Implement the first problem in a module called `date.py`, the second problem in a module called `datefuns.py`, and the third one in a module called `poly.py`. *Your name and RUID should appear as a comment at the very top of each module.* Points will be deducted if you do not follow the specified naming convention.

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

1. Go to the "Assignments" tab of the Canvas site for this course.
2. Click on "Programming Assignment #4" under Homework Assignments.
3. Upload your homework files (`date.py`, `datefuns.py` and `poly.py`) when you are ready to submit.

You must submit your assignment at or before 11:59PM on November 8, 2020.