CS-UY 1134: Data Structures and Algorithms
Lab 2: A Polynomial Class

Friday, June 7, 2019

1 Instructions

This lab will cover some fundamentals of Python programming and basic object-oriented programming. You will need to be familiar with how to write a class in Python, as well as operator overloading.

Try to write a completed implementation by the end of the lab time. It is a good idea to think carefully about how to solve the problem and plan your approach carefully. You do not (and probably *should not*) begin writing code immediately. It is good practice to work with pen and paper, working out some sample inputs and outputs by hand, and possibly drawing diagrams.

As added practice, it can be a good idea to try writing your code out on pen and paper first. In addition to being a useful approach for planning out a solution, it is good practice for exams, where you will have to write some code on paper.

When you type up your code, write some test cases. Try multiple test cases for each function you have to implement, and consider edge cases carefully.

1.1 What to submit

If you believe you have a working solution, have a TA or the professor check your solution.

If you don't finish by the end of the lab, you will may submit a solution on NYU Classes by Sunday June 9, 11:55 PM.

Either when a TA has checked your code, or by the Sunday submission deadline, submit a single ".py" file to the "Lab 1: Polynomial" assignment on NYU Classes. The ".py" file should contain your implementation of the Polynomial class described below. Place any test code you wrote in a "main" function.

2 The Polynomial Class

For this lab, we wish to implement a class to represent a polynomial. We will represent a polynomial by storing a list with its coefficients, as a data member of

the class. The first element of the list (index 0) will represent the constant; the second (index 1) will represent the coefficient of the x term; and so on, with each next element representing the coefficient of the next power of the polynomial.

Example. The coefficient list of $p(x) = 5x^3 - 17x + 42$ would be `[42,-17,0,5]`.

***Note** that the length of the coefficient list is one more than the degree of the polynomial. (Recall that the* degree *is the largest power with a non-zero coefficient.)*

The class you implement should include the following methods:

- **Constructor**: The constructor should take in a list of coefficients and initialize a polynomial with the coefficients given in the list. If no list is given, initialize the polynomial $p(x) = 0$.

- **repr**: You should implement the `__repr__` special method to return a string representation of the polynomial. You may use `^` instead of superscript to represent powers.

- **eval**: This method should take a value and evaluate the polynomial for that value. For example, with the polynomial in the example above (i.e., $p(x) = 5x^3 - 17x + 42$), calling `eval(1)` should return 30.

- **Addition**: You should implement the addition operator (using operator overloading) to add two polynomials. Recall that adding two polynomials means adding the coefficients of the same power. Beware of cases where the two polynomials being added have different degrees. For example: $(3x^4 - 10x^2 + 5x + 7) + (2x^{12} + 3x^2 - 2) = 2x^{12} + 3x^4 - 7x^2 + 5$.

- **Multiplication**: You should implement the multiplication operator (using operator overloading) to multiply two polynomials. In order to multiply polynomials, you will have to multiply each pair of terms — each pair of terms when multiplied gives a new term whose coefficient is the product of the two terms, and whose power is the sum of the powers of the two terms (e.g., $2x^4 * 7x^3 = 14x^7$) — and group terms of the same power.

  A simple example: $(x + 1) * (2x + 3) = 2x^2 + 5x + 3$.

  And a more complicated example: $(3x^3 + 2x) * (2x^{11} + 5x^3 + x) = 6x^{14} + 15x^6 + 3x^4 + 4x^{12} + 10x^4 + 2x^2 = 6x^{14} + 4x^{12} + 15x^6 + 13x^4 + 2x^2$.

- A function **polySequence** which takes a `start`, `end`, and `step`, and returns a *generator*. The generator will evaluate the polynomial for the value `start`, then `start` + `end`, and so on, up to `end` and yield these values one at a time. If no `step` is given, a step of 1 should be used (similar to the `range` method). As an example, if `p` represents the polynomial $2x + 1$ (coefficient list `[1,2]`), the code

  ```
  for  val in p.polySequence(0,5):
       print(val)
  ```

  should print the values 1, 3, 5, 7, and 9 on separate lines.

* Extra Challenge: Derivative

For an extra challenge, write a *list comprehension* to construct the coefficient list of the *derivative* of a given polynomial. Use this list comprehension to implement a new member method, **derivative** that computes the derivative and returns it as a *new* Polynomial object.

Note: The derivative of a polynomial is computed by using the *power rule*: the derivative of $p(x) = x^n$ is nx^{n-1} (for any number n), and you can apply this rule to each term of a polynomial. For example, the derivative of $2x^3 + 4x + 1$ is $6x^2 + 4$.