3.C Multiply by Term

Input:

poly, an instance of Polynomial that this method is acting on

coefficient, an integer number which is the coefficient of the multiplied monomial

power, an integer value which is the degree of the multiplied term

Output: a newly formed Polynomial which is the product of the polynomial and the monomial

Method called: get_terms

```
1. new_terms ← a new instance of the Polynomial formed with the arguments of the map
   returned by calling get_terms of poly
2. new_polynomial ← an empty map
3. For each exponent and its corresponding mapped new_coefficient in new_terms, do
    A. $new\_polynomial_{exoonet+power}$ ← new_coefficent*coefficient in z256
4. Return a new Polynomial that is constructed with the argument new_polynomial
```

4.A Polynomial Addition

Inputs:

other_polynomial, a *Polynomial* instance

poly, an instance of *Polynomial* that this method is acting on

Outputs:

a new *Polynomial* that is the sum of both *Polynomials*

Method called: *get_terms*, *add_term*

```
1. result←Polynomial instance that is a copy of poly
2. For each power and its respective mapped coefficient in the map returned by get_terms
   on other_polynomial, do
   A. result←Polynomial returned by calling add_term with arguments coefficient and
   power on result
3. Return result
```

4.C Polynomial Multiplication

Inputs:

other_polynomial, a *Polynomial* instance

poly, an instance of *Polynomial* that this method is acting on

Output:

a new *Polynomial* that is the product of both *Polynomials*.

Methods called: *get_terms*, *multiply_by_term*, *add_polynomial*

```
1. result← an instance of a Polynomial instance with no arguments
2. For each power and its respective mapped coefficient in the map returned by get_terms
      on other_polynomial, do
      A.term_product←Polynomial returned by multiply_by_term with arguments coefficient
        and power on poly
      B.result←Polynomial returned by add_polynomial with argument term_product on poly
3. Return result
```

4.D Polynomial Remainder

Inputs:

poly, an instance of *Polynomial* that this method is acting on

denominator, a *Polynomial* object

Outputs: a new *Polynomial* that is the remainder

Methods called: *get_terms*, *get_degree, subtract_polynomial*, *multiply_by_term*

Function used: *divide_terms*

```
1.new_poly← an instance of Polynomial constructed using the argument of the map
returned by get_terms on poly
2. degree←integer returned by get_degree on new_poly
3. If degree=0 and the integer of get_degree on denominator is equal to 0, do
        A.Return a Polynomial instance constructed with a map of 0←0 as an argument
4. While degree is greater than or equal to the integer of get_degree on denominator and
degree is not equal to 0, do
        A. max_degree← the integer of get_degree on new_poly
        B. max_coefficient← the integer of get_coefficient with argument max_degree on
        new_poly
        C. max_degree2← the integer of get_degree on denominator
        D. max_coefficient2← the integer of get_coefficient with argument max_degree2 on
        denominator
        E. quotient← the Polynomial of function divide_terms with arguments
        max_coefficient, max_degree, max_coefficient2, max_degree2
        F.product←the Polynomial returned by multiply_by_polynomial with argument
        denominator on quotient
        G.new_poly← the Polynomial returned by subtract_polynomial with argument
        product on new_poly
        H.degree←integer returned by get_degree on new_poly
5. Return new_poly
```

Discussion

1. If you are given a message that you want to encode and a value, which indicates how many error correction bytes you need, is it possible to guarantee that you will not have any coefficients that are equal to zero in the remainder from dividing the message polynomial by the generator polynomial? If there were coefficients that are equal to zero in the encoded data, would it be a problem? Why or why not?

No, it is not possible to guarantee that all the coefficients in the remainder from dividing the message polynomial by the generator polynomial will be nonzero; sometimes, coefficients can be zero depending on the message and the generator polynomial chosen. Having zero coefficients in the encoded data is not inherently a problem, since zero is just as valid a symbol as any other in the code's alphabet. The key property for error correction is the structure and position of the coefficients, rather than the fact that they are all nonzero.

2. We have discussed the importance of modularity and writing your recipes/code in such a way that you can reuse them. If you needed a `Polynomial` class to represent polynomials with regular, real-number coefficients (as opposed to coefficients that are elements of), how could you minimally change the code you have already written in order to reuse it for this purpose?

To reuse a Polynomial class for real-number coefficients, you could minimally change the type of the coefficients' storage from, for example, a finite field or integers to floating-point or double types, so that the class operates on real numbers. This means updating type definitions and any arithmetic methods to work correctly with real numbers, without changing the main interface or polynomial manipulation logic.