Polynomial

- Total points: 11 (divided among 9 tasks)
- Number of submissions allowed: 6
- General rules are as usual: the `"ex.py"` file should be clean; no unnecessary printing; only the behavior matters; refer to the `"runtest.py"` file for specifics and examples. The full specification is at the end of this document.

Introduction

We want to write a class called `Polynomial`, whose objects represent polynomials in one variable. The idea is that in order to store a polynomial $a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ we can just store the list of coefficients $(a_i)_{i=0...n}$. We will call this list `coeffs`.

For example, for $2 + 4x + 6x^2 + x^4$, the list is `coeffs=[2,4,6,0,1]`.

Notice that:

- The list is ordered from small to high degree: `coeffs[i]` corresponds to a_i, the coefficient of the monomial x^i.
- There may be zeros in the list, corresponding to missing monomials (in the above example the monomial x^3 is missing). Trailing zeros do not affect which polynomial we are representing: for example the list `coeffs=[2,3]` represents the polynomial $2x^0 + 3x^1 = 2 + 3x$, and `coeffs=[2,3,0]` also represents $2x^0 + 3x^1 + 0x^2 = 2 + 3x$, and so does `[2,3,0,0]` etc.
- For a polynomial of degree n we need a list of length at least `n+1`. For example $1 + x^3$ has degree 3 and its minimal representation is `coeffs=[1,0,0,1]` of length `4`. As a special case, the zero polynomial $0x^0$ is considered to have degree 0, but it can be represented with `coeffs=[]` (thus, with a list of the same length as the degree), as well as with `[0]`, `[0,0]`, `[0,0,0]` etc.

Thus our `Polynomial` objects will only store a single attribute, `coeffs`. Everything else (e.g. polynomial evaluation, addition etc.) will be implemented via appropriate methods. For simplicity we will restrict our polynomials to having integer coefficients.

Tasks

Each task is worth 1 or 1.5 points. The test file `"runtest.py"` contains code that tests all tasks independently. For each task, if it fails you will see a backtrace telling you where the error was produced, otherwise it prints a “tests ok” message (meaning that the code *may* be correct).

In the initial file, the class `Polynomial` is defined and a few methods are already written: the constructor, a method for testing equality, and one for representing objects as strings. These are all basically working, but you’ll need to improve them.

1. **[1.5 points]** The constructor takes an `iterable` argument and stores it in the `coeffs` attribute. You must provide a default of `None` to this argument, and then in the constructor you must detect when the argument is `None`. The result must be that one can write `Polynomial()` or `Polynomial(None)` and obtain the same effect as passing an empty iterable, like e.g. `Polynomial([])` or `Polynomial(())`.

2. **[1.5 points]** After the constructor has created the `coeffs` attribute, add checks to make sure that each element of the list is integer, and raise a `TypeError` otherwise. (Note: an empty list is valid, don’t raise any errors in that case.)

3. **[1.5 points]** Write an `evaluate` method that takes one argument `x` (assumed to be a number, no checks required) and evaluates the polynomial at that value. For example, if `coeffs==[0, 3, 1]`, representing $0 + 3x + x^2$, and `x==2` the result should be `0+6+4=10`.

4. **[1.5 points]** The code for comparing two `Polynomial` objects with `==` is already written. Add a check so that if the second term of the comparison is not a `Polynomial` the comparison operation returns `False` instead of crashing. However, this should not be the result of directly returning `False`; instead you need to return a special value that ends up having that effect.

5. **[1 point]** Write a method that allows to write `p * k` where `p` is a `Polynomial` and `k` is an integer. The result should be a new `Polynomial`. The `coeffs` list of that new object should have the same length as that of `p`, with the entries

multiplied by `k`. If the second argument is not an `int`, the operation should fail, not by explicitly raising an exception but rather by returning a special value. Then write also the reflected method that allows to write `k * p`, i.e. the same multiplication in reverse order. This method must rely on the previous one (but without explicit method calls) and consist of a single line of code.

6. **[1 point]** Write a `degree` method that returns the degree of the polynomial. The degree is the index of the last non-zero element of `coeffs`, except in the special case of an empty list (which represents the zero polynomial as explained above).

7. **[1 point]** Write a `normalize` method that removes all the trailing zeros in the `coeffs` list. For example, if the list is `[1,0,2,2,0,0]` it must become `[1,0,2,2]`; if it is `[0,0,0]` it must become `[]`; if it is `[3,4]` it must remain unchanged. The deletion must occur in-place: the identity of the list must not change. You may choose to use the previous `degree` method for this. The method returns `None`.

8. **[1 point]** Improve the method for representing polynomials as strings. Consider for example the polynomial that you would normally write as $2 + 4x + 6x^2 + x^4$, and which may be stored internally as `[2,4,6,0,1,0]`. With the initial code, it is represented as `"2x^0+4x^1+6x^2+0x^3+1x^4+0x^5"`. We want to represent it instead as `"2+4x+6x^2+x^4"`. Furthermore, negative numbers are treated incorrectly, e.g. `[-2,-1,-3]` is represented as `"-2x^0+-1x^1+-3x^2"`. We need to add parentheses around negative coefficients, e.g. `"(-2)+(-1)x+(-3)x^2"`.

   Specifically, the new format rules are:

   - Monomials with coefficient `0` should be skipped. There's an exception, which is basically already implemented though: the zero polynomial, which may have associated lists `[]`, `[0]`, `[0,0]` etc., should be represented as `"0"`, not as `""`.
   - Every negative coefficient should be written in parentheses.
   - The `x^0` should always be omitted, and `x^1` should be written as `x` instead.
   - Except for degree `0` monomials, a coefficient of `1` should be omitted. E.g. `[1,1,1]` should be represented as `"1+x+x^2"`.

   Hint: build the coefficient part of the string and the `"x^i"` part of the string separately

9. **[1 point]** Write a method that allows to write `p + q` where `p` and `q` are two `Polynomial`s. The result should be a new `Polynomial`. The `coeffs` list of that new object should have the same length as the maximum length of the arguments, and the coefficients should be the sum of corresponding coefficients (if one of the input lists is shorter the missing coefficients are considered to be zero). If the second argument is not a `Polynomial`, the operation should fail, not by throwing an exception but rather by returning a special value.

General rules

In all methods, you can use whatever constructs you want. Only the behavior matters, as long as you follow the indications.

The class methods must not print anything. When the file `"ex.py"` is loaded nothing should be printed. Do not write anything else in the file, just the class(es) and, if needed, `import` statements.

You can do tests by typing `python runtest.py` in the terminal, or pressing the `RUN` button. Edit the file `"runtest.py"` as much as you want when you test. That file is for your own use and it's not used when grading.

Remember that copies of the original files are always present in the `"startercode"` directory, in case you need them.

As for examples of how everything should work, refer to the test script.

Extra challenges (proposals)

- Implement a derivative operation. E.g. if the polynomial is `[3, 4, 5, 6]` the derivative is `[4, 10, 18]`. Watch out for the corner case.

- Extend the multiplication method to allow multiplying two polymonials.

- Enforce normalization in the constructor (you can still use the `normalize` method, maybe change its name to `_normalize` since it wouldn't be part of the interface any more). Then in the rest of the code assume that all polynomials are always normalized.

- In the printing part, instead of putting negative coefficients in parentheses, omit the `+` sign (but not in the zero-degree monomial). For example, with coefficient list `[-2,1,-3,4,-5]` you want the output string to be `"-2+x-3x^2+4x^3-5x^4"`. You may even manage to make them print more nicely with a bit of Unicode, as `-2+x-3x²+4x³-5x⁴"`, but make sure that you can represent terms of any degree! E.g. `1+2x¹⁵`.

- You can try to implement a parser: if the input argument is a string, try to split it and read out the coefficients from it, as in `"1+x^3"` → `[1,0,3]` (feel free to decide the format of the input string).
- Allow polynomials to use rational coefficients, and to be evaluated at rational values. You can use `from fractions import Fraction`, it's a built-in Python module, you can look it up in the docs. Or you can implement the class yourself, like this: store the numerator and denominator in reduced form (use the `math.gcd` method for this in the constructor; give an error in case of zero denominator; ensure that the denominator is always positive). Implement addition, subtratction, multiplication, division (both forms), (integer) power, unary plus and minus. Allow all the binary operations `+`, `-`, `*`, `/`, `//` to be performed with integers as well. Give reflected versions too. Also implement printing in a nice form, e.g. as `-3/4`.
- Implement division between Polynomials with rational coefficients, returning two polynomials, the quotient and the remainder, in normalized form.
- Implement polynomial interpolation: given `n` points (with rational coordinates), return a polynomial of degree at most `n-1` that passes through those points.