420-LCW-05 Programming Techniques and Applications - Assignment 1

February 16, 2018

This is our first regular assignment in the course. To start, we'll mention some general requirements that will apply to all of these assignments:

1. **Identification section** Do this for *every* Python file in every assignment in this course. This section must be either in a comment, with a '#' preceding each line, or enclosed within triple quotes (''). The grader and I need this section for the *accurate processing of your assignment*. Assignments missing this may lose up to 5% of the total mark.

   Example:

   ```
   """
   Justin Trudeau, 1234567
   Sunday, February 31
   R. Vincent, instructor
   Assignment 1
   """
   ```

   Obviously substitute your name, Marianopolis ID, and the correct date for the appropriate fields!

2. Always include additional comments with your code. These need not explain every individual line of your program, but consider using comments for the following situations:

   - A brief explanation of a particular variable's purpose, included on the first line where the variable is defined, e.g.:

     ```
     hi = 100 # Define the upper limit of the range.
     ```

   - A note mentioning any website or person you may have consulted with to help with the assignment.
   - A comment describing any constant value that appears in your code.

   In addition, each `def` statement, whether for a global function or class method should include at least a brief docstring.

3. Your submission for assignment will typically include multiple Python files, which have the extension `.py`. Before submission, these files must be combined into a single ZIP archive file (extension `.zip`). If you do not know how to create a ZIP file, I will demonstrate this in lab.

4. Be sure to respect other instructions specified in the assignment. Part of each assignment is to correctly follow the instructions as closely as possible.

Exercise 1 - Classes

In this part, we will review object-oriented programming in Python by creating a class to perform operations on *polynomials*.

In mathematics a polynomial is represented by its coefficients in increasing order of degree. Abstractly, it is a finite ordered sequence of numbers all chosen from the same base-set, e.g. integers, reals or complex numbers.

In this implementation, the coefficients are stored in a Python list or tuple, with the zero-index element holding the constant term. The length of the list equals the degree of the polynomial plus one. If a term of intermediate degree is missing, we represent it with a zero coefficient.

Example The list `[1, 2, 3]` represents the polynomial $1 + 2x + 3x^2$.

1. Create your class named `polynomial` in a file named `polynomial.py`. At a minimum, your class should support the following methods:

   - `__init__(self, iterable = None)` - initialize the polynomial. If the iterable is `None`, the polynomial is initialized to be the "zero polynomial" (see below). Provide for this case by defining the "zero polynomial" as having an empty coefficient list and a negative degree, say -1. (In mathematics the identically zero polynomial is defined as having the degree minus infinity!) *Be sure that your methods all work with a zero polynomial*! Hint: remove any zero-valued entries from the end of the coefficient list before returning from the `__init__()` method.
   - `degree(self)` - return an integer representing the degree of the polynomial.

- `__str__(self)` - return a string representation of the polynomial. For the example given above, the string would be "1 + 2x^1 + 3x^2".
- `__eq__(self, poly)` - compare two polynomials for equality. Be sure your code handles the case where `poly` is `None`!

2. Once you have the basic implementation working, write a new method, `__add__(self, poly)`. This should allow you to add two polynomial objects using the Python addition operator. Remember that the return value of this method will be a new polynomial (do *not* modify the parameters), so your method will have to construct the resulting polynomial.

   Do NOT assume that the two polynomials have equal degree.

3. Create some client code for your class in another file named `polytest.py`. In this module, "exercise" your class including the cases when, during addition, some coefficient pairs cancel out, and also the case when the sum is the zero polynomial.

   When the client program (i.e. the user of your class) uses a `polynomial` object, it must not âĂIJknowâĂİ the details of the internal representation. This is called *encapsulation*, needed to make a `polynomial` into an ADT (Abstract Data Type). This might come in handy in a future version, where we might want change the internal representation of the polynomial into a linked list, for example.

4. **Optional** Add a method `evaluate(self, x)` to evaluate a polynomial at a specific value `x`. For an additional challenge, do this efficiently by looking up Horner's rule and using that algorithm.

5. **Optional** Add a method `__mul__(self, poly)` to allow for multiplication of polynomials using the Python multiplication operator.

Exercise 2 - Stacks

In this exercise, we will write a very simple syntax checker for Python. All our program will do is read a Python file and tell us whether or not the parentheses, square brackets, and curly braces are all properly matched. Since we don't want to get too complicated, we will not worry about recognizing strings or comments - we'll assume that any curly brace, parenthesis, or square bracket we encounter is part of the program.

You'll want to use a stack to implement this. We have included the simple stack class shown in lecture that is derived from a Python list.

Your program should prompt for the name of a file, open and read the file, then run the syntax checking.

To perform the syntax checking, you will need to push each "left" bracket character onto a stack. Whenever you encounter a "right" bracket, you will check the stack and see if the top is the appropriate corresponding type. If the pair matches, pop the stack.

Your program should detect each of the following situations:

1. Unclosed (missing) brackets, e.g. `(1 + (2 * 3)`
2. Mismatched brackets, e.g. `(0, 1, 2]`
3. Extra brackets, e.g. `1 + (2 * 3))`

Submitting your work

When you have finished both sections, combine all of your files into a single ZIP file and upload that to Omnivox.