CMPSC 442: Homework 1 [100 points]

Release Date	Monday, August 26, 2019, 12:00 am
Due Date	Sunday, September 1, 2019, 11:59 pm

TO SUBMIT HOMEWORK

To submit homework for a given homework assignment:

1. You *must* download the homework template file from Canvas, located in Files/Homework Templates, and modify this file to complete your homework. Each template file is a python file that will give you a headstart in creating your homework python script. For a given homework number N, the template file name is homeworkN-cmpsc442.py. For example, the template for homework #1 is homework1-compsc442.py. IF YOU DO NOT USE THE CORRECT TEMPLATE FILE, YOUR HOMEWORK CANNOT BE GRADED AND YOU WILL RECEIVE A ZERO.

2. You *must* rename the file by replacing the file root using your PSU id that consists of your initials followed by digits. This is the same as the part of your PSU email that precedes the "@" sign. For example, your instructor's email is rjp49@cse.psu.edu, and her PSU id is rjp49. Your homework files for every assignment will have the same name, e.g., rjp49.py. IF YOU DO NOT RENAME YOUR HOMEWORK FILE CORRECTLY, IT WILL NOT BE GRADED AND YOU WILL RECEIVE A ZERO.

3. You *must* upload your homework to the correct assignments area in Canvas by 11:59 pm on the due date. IF YOU DO NOT UPLOAD YOUR HOMEWORK TO THE CORRECT LOCATION, IT WILL NOT BE GRADED AND YOU WILL RECEIVE A ZERO.

Instructions

In this assignment, you will answer some conceptual questions about Python and write a collection of basic algorithms and data structures.

A skeleton file `homework1-cmpsc442.py` containing empty definitions for each question has been provided. Since portions of this assignment will be graded automatically, none of the names or function signatures in this file should be modified. However, you are free to introduce additional variables or functions if needed.

Unless explicitly stated otherwise, you may not import any of the standard Python modules, meaning your solutions should not include any lines of the form `import x` or `from x import y`. Accordingly, you may find it helpful to refresh yourself on Python's built-in functions and data types.

You will find that in addition to a problem specification, each programming question also includes a

pair of examples from the Python interpreter. These are meant to illustrate typical use cases to clarify the assignment, and are not comprehensive test suites.

You are strongly encouraged to follow the Python style guidelines set forth in PEP 8, which was written in part by the creator of Python. However, your code will not be graded for style.

Once you have completed the assignment, you should submit your file in Canvas.

1. Python Concepts (6 points) [6 points]

For each of the following questions, write your answers as triply-quoted strings using the indicated variables in the provided file.

1. **[2 points]** [2 points] Explain what it means for Python to be both strongly and dynamically typed, and give a concrete example of each.

2. **[2 points]** [2 points] You would like to create a dictionary that maps some 2-dimensional points to their associated names. As a first attempt, you write the following code:

   ```
   points_to_names = {[0, 0]: "home", [1, 2]: "school", [-1, 1]: "market"}
   ```

   However, this results in a type error. Describe what the problem is, and propose a solution.

3. **[2 points]** [2 points] Consider the following two functions, each of which concatenates a list of strings.

   ```
   def concatenate1(strings):
       result = ""
       for s in strings:
           result += s
       return result
   ```

   ```
   def concatenate2(strings):
       return "".join(strings)
   ```

   One of these approaches is significantly faster than the other for large inputs. Which version is better, and what is the reason for the discrepancy?

2. Working with Lists (15 points) [15 points]

1. **[5 points]** [5 points] Consider the function `extract_and_apply(l, p, f)` shown below, which extracts the elements of a list `l` satisfying a boolean predicate `p`, applies a function `f` to each such element, and returns the result.

   ```
   def extract_and_apply(l, p, f):
       result = []
       for x in l:
           if p(x):
   ```

```
            result.append(f(x))
    return result
```

Rewrite `extract_and_apply(l, p, f)` in one line using a list comprehension.

2. **[5 points]** [5 points] Write a function `concatenate(seqs)` that returns a list containing the concatenation of the elements of the input sequences. Your implementation should consist of a single list comprehension, and should not exceed one line.

```
>>> concatenate([[1, 2], [3, 4]])
[1, 2, 3, 4]
```

```
>>> concatenate(["abc", (0, [0])])
['a', 'b', 'c', 0, [0]]
```

3. **[5 points]** [5 points] Write a function `transpose(matrix)` that returns the transpose of the input matrix, which is represented as a list of lists. Recall that the transpose of a matrix is obtained by swapping its rows with its columns. More concretely, the equality `matrix[i][j] == transpose(matrix)[j][i]` should hold for all valid indices `i` and `j`. You may assume that the input matrix is well-formed, i.e., that each row is of equal length. You may further assume that the input matrix is non-empty. Your function should not modify the input.

```
>>> transpose([[1, 2, 3]])
[[1], [2], [3]]
```

```
>>> transpose([[1, 2], [3, 4], [5, 6]])
[[1, 3, 5], [2, 4, 6]]
```

3. Sequence Slicing (6 points) [6 points]

The functions in this section should be implemented using sequence slices. Recall that the slice parameters take on sensible default values when omitted. In some cases, it may be necessary to use the optional third parameter to specify a step size.

1. **[2 points]** [2 points] Write a function `copy(seq)` that returns a new sequence containing the same elements as the input sequence.

```
>>> copy("abc")
'abc'
>>> copy((1, 2, 3))
(1, 2, 3)
```

```
>>> x = [0, 0, 0]; y = copy(x)
>>> print x, y; x[0] = 1; print x, y
[0, 0, 0] [0, 0, 0]
[1, 0, 0] [0, 0, 0]
```

2. **[2 points]** [2 points] Write a function `all_but_last(seq)` that returns a new sequence containing all but the last element of the input sequence. If the input sequence is empty, a new empty sequence of the same type should be returned.

```
>>> all_but_last("abc")
```

```
>>> all_but_last("")
```

```
'ab'
>>> all_but_last((1, 2, 3))
(1, 2)
```

```
''
>>> all_but_last([])
[]
```

3. **[2 points]** [2 points] Write a function `every_other(seq)` that returns a new sequence containing every other element of the input sequence, starting with the first. *Hint: This function can be written in one line using the optional third parameter of the slice notation.*

```
>>> every_other([1, 2, 3, 4, 5])
[1, 3, 5]
>>> every_other("abcde")
'ace'
```

```
>>> every_other([1, 2, 3, 4, 5, 6])
[1, 3, 5]
>>> every_other("abcdef")
'ace'
```

4. Combinatorial Algorithms (11 points) [11 points]

The functions in this section should be implemented as generators. You may generate the output in any order you find convenient, as long as the correct elements are produced. However, in some cases, you may find that the order of the example output hints at a possible implementation.

Although generators in Python can be used in a variety of ways, you will not need to use any of their more sophisticated features here. Simply keep in mind that where you might normally `return` a list of elements, you should instead `yield` the individual elements.

Since the contents of a generator cannot be viewed without employing some form of iteration, we wrap all function calls in this section's examples with the `list` function for convenience.

1. **[6 points]** [6 points] The prefixes of a sequence include the empty sequence, the first element, the first two elements, etc., up to and including the full sequence itself. Similarly, the suffixes of a sequence include the empty sequence, the last element, the last two elements, etc., up to and including the full sequence itself. Write a pair of functions `prefixes(seq)` and `suffixes(seq)` that yield all prefixes and suffixes of the input sequence.

```
>>> list(prefixes([1, 2, 3]))
[[], [1], [1, 2], [1, 2, 3]]
>>> list(suffixes([1, 2, 3]))
[[1, 2, 3], [2, 3], [3], []]
```

```
>>> list(prefixes("abc"))
['', 'a', 'ab', 'abc']
>>> list(suffixes("abc"))
['abc', 'bc', 'c', '']
```

2. **[5 points]** [5 points] Write a function `slices(seq)` that yields all non-empty slices of the input sequence.

```
>>> list(slices([1, 2, 3]))
[[1], [1, 2], [1, 2, 3], [2], [2, 3],
```

```
>>> list(slices("abc"))
['a', 'ab', 'abc', 'b', 'bc', 'c']
```

```
[3]]
```

5. Text Processing (20 points) [20 points]

1. **[5 points]** [5 points] A common preprocessing step in many natural language processing tasks is text normalization, wherein words are converted to lowercase, extraneous whitespace is removed, etc. Write a function `normalize(text)` that returns a normalized version of the input string, in which all words have been converted to lowercase and are separated by a single space. No leading or trailing whitespace should be present in the output.

```
>>> normalize("This is an example.")
'this is an example.'
```

```
>>> normalize("   EXTRA   SPACE   ")
'extra space'
```

2. **[5 points]** [5 points] Write a function `no_vowels(text)` that removes all vowels from the input string and returns the result. For the purposes of this problem, the letter 'y' is not considered to be a vowel.

```
>>> no_vowels("This Is An Example.")
'Ths s n xmpl.'
```

```
>>> no_vowels("We love Python!")
'W lv Pythn!'
```

3. **[5 points]** [5 points] Write a function `digits_to_words(text)` that extracts all digits from the input string, spells them out as lowercase English words, and returns a new string in which they are each separated by a single space. If the input string contains no digits, then an empty string should be returned.

```
>>> digits_to_words("Zip Code: 19104")
'one nine one zero four'
```

```
>>> digits_to_words("Pi is 3.1415...")
'three one four one five'
```

4. **[5 points]** [5 points] Although there exist many naming conventions in computer programming, two of them are particularly widespread. In the first, words in a variable name are separated using underscores. In the second, words in a variable name are written in mixed case, and are strung together without a delimiter. By mixed case, we mean that the first word is written in lowercase, and that subsequent words have a capital first letter. Write a function `to_mixed_case(name)` that converts a variable name from the former convention to the latter. Leading and trailing underscores should be ignored. If the variable name consists solely of underscores, then an empty string should be returned.

```
>>> to_mixed_case("to_mixed_case")
'toMixedCase'
```

```
>>> to_mixed_case("__EXAMPLE__NAME__")
'exampleName'
```

6. Polynomials (37 points) [37 points]

In this section, you will implement a simple `Polynomial` class supporting basic arithmetic, simplification, evaluation, and pretty-printing. An example demonstrating these capabilities is shown below.

```
>>> p, q = Polynomial([(2, 1), (1, 0)]), Polynomial([(2, 1), (-1, 0)])
>>> print p; print q
2x + 1
2x - 1
>>> r = (p * p) + (q * q) - (p * q); print r
4x^2 + 2x + 2x + 1 + 4x^2 - 2x - 2x + 1 - 4x^2 + 2x - 2x + 1
>>> r.simplify(); print r
4x^2 + 3
>>> [(x, r(x)) for x in range(-4, 5)]
[(-4, 67), (-3, 39), (-2, 19), (-1, 7), (0, 3), (1, 7), (2, 19), (3, 39), (4, 67)]
```

1. **[2 points]** [2 points] In this problem, we will think of a polynomial as an immutable object, represented internally as a tuple of coefficient-power pairs. For instance, the polynomial $2x + 1$ would be represented internally by the tuple `((2, 1), (1, 0))`. Write an initialization method `__init__(self, polynomial)` that converts the input sequence `polynomial` of coefficient-power pairs into a tuple and saves it for future use. Also write a corresponding method `get_polynomial(self)` that returns this internal representation.

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> p.get_polynomial()
((2, 1), (1, 0))
```

```
>>> p = Polynomial(((2, 1), (1, 0)))
>>> p.get_polynomial()
((2, 1), (1, 0))
```

2. **[4 points]** [4 points] Write a `__neg__(self)` method that returns a new polynomial equal to the negation of `self`. This method will be used by Python for unary negation.

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> q = -p; q.get_polynomial()
((-2, 1), (-1, 0))
```

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> q = -(-p); q.get_polynomial()
((2, 1), (1, 0))
```

3. **[3 points]** [3 points] Write an `__add__(self, other)` method that returns a new polynomial equal to the sum of `self` and `other`. This method will be used by Python for addition. No simplification should be performed on the result.

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> q = p + p; q.get_polynomial()
```

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> q = Polynomial([(4, 3), (3, 2)])
```

```
((2, 1), (1, 0), (2, 1), (1, 0))
```

```
>>> r = p + q; r.get_polynomial()
((2, 1), (1, 0), (4, 3), (3, 2))
```

4. **[3 points]** [3 points] Write a `__sub__(self, other)` method that returns a new polynomial equal to the difference between `self` and `other`. This method will be used by Python for subtraction. No simplification should be performed on the result.

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> q = p - p; q.get_polynomial()
((2, 1), (1, 0), (-2, 1), (-1, 0))
```

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> q = Polynomial([(4, 3), (3, 2)])
>>> r = p - q; r.get_polynomial()
((2, 1), (1, 0), (-4, 3), (-3, 2))
```

5. **[5 points]** [5 points] Write a `__mul__(self, other)` method that returns a new polynomial equal to the product of `self` and `other`. This method will be used by Python for multiplication. No simplification should be performed on the result. Your result does not need to match the examples below exactly, as long as the same terms are present in some order.

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> q = p * p; q.get_polynomial()
((4, 2), (2, 1), (2, 1), (1, 0))
```

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> q = Polynomial([(4, 3), (3, 2)])
>>> r = p * q; r.get_polynomial()
((8, 4), (6, 3), (4, 3), (3, 2))
```

6. **[3 points]** [3 points] Write a `__call__(self, x)` method that returns the result of evaluating the current polynomial at the point `x`. This method will be used by Python when a polynomial is called as a function. *Hint: This method can be written in one line using Python's exponentiation operator, `**`, and the built-in `sum` function.*

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> [p(x) for x in range(5)]
[1, 3, 5, 7, 9]
```

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> q = -(p * p) + p
>>> [q(x) for x in range(5)]
[0, -6, -20, -42, -72]
```

7. **[8 points]** [8 points] Write a `simplify(self)` method that replaces the polynomial's internal representation with an equivalent, simplified representation. Unlike the previous methods, `simplify(self)` does not return a new polynomial, but rather acts in place. However, because the fundamental character of the polynomial is not changing, we do not consider this to violate the notion that polynomials are immutable.

   The simplification process should begin by combining terms with a common power. Then, terms with a coefficient of zero should be removed, and the remaining terms should be sorted in decreasing order based on their power. In the event that all terms have a coefficient of zero

after the first step, the polynomial should be simplified to the single term $0 \cdot x^0$, i.e. `(0, 0)`.

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> q = -p + (p * p); q.get_polynomial()
((-2, 1), (-1, 0), (4, 2), (2, 1),
 (2, 1), (1, 0))
>>> q.simplify(); q.get_polynomial()
((4, 2), (2, 1))
```

```
>>> p = Polynomial([(2, 1), (1, 0)])
>>> q = p - p; q.get_polynomial()
((2, 1), (1, 0), (-2, 1), (-1, 0))
>>> q.simplify(); q.get_polynomial()
((0, 0),)
```

8. **[9 points]** [9 points] Write a `__str__(self)` method that returns a human-readable string representing the polynomial. This method will be used by Python when the `str` function is called on a polynomial, or when a polynomial is printed.

   In general, your function should render polynomials as a sequence of signs and terms each separated by a single space, i.e. `"sign1 term1 sign2 term2 ... signN termN"`, where signs can be `"+"` or `"-"`, and terms have the form `"ax^b"` for coefficient `a` and power `b`. However, in adherence with conventional mathematical notation, there are a few exceptional cases that require special treatment:

   - The first sign should not be separated from the first term by a space, and should be left blank if the first term has a positive coefficient.
   - The variable and power portions of a term should be omitted if the power is *0*, leaving only the coefficient.
   - The power portion of a term should be omitted if the power is *1*.
   - Coefficients with magnitude *0* should always have a positive sign.
   - Coefficients with magnitude *1* should be omitted, unless the power is *0*.

   You may assume that all polynomials have integer coefficients and non-negative integer powers.

```
>>> p = Polynomial([(1, 1), (1, 0)])
>>> qs = (p, p + p, -p, -p - p, p * p)
>>> for q in qs: q.simplify(); str(q)
...
'x + 1'
'2x + 2'
'-x - 1'
'-2x - 2'
'x^2 + 2x + 1'
```

```
>>> p = Polynomial([(0, 1), (2, 3)])
>>> str(p); str(p * p); str(-p * p)
'0x + 2x^3'
'0x^2 + 0x^4 + 0x^4 + 4x^6'
'0x^2 + 0x^4 + 0x^4 - 4x^6'
>>> q = Polynomial([(1, 1), (2, 3)])
>>> str(q); str(q * q); str(-q * q)
'x + 2x^3'
'x^2 + 2x^4 + 2x^4 + 4x^6'
'-x^2 - 2x^4 - 2x^4 - 4x^6'
```

7. Feedback (5 points) [5 points]

1. **[1 point]** [1 point] Approximately how long did you spend on this assignment?
2. **[2 points]** [2 points] Which aspects of this assignment did you find most challenging? Were there any significant stumbling blocks?
3. **[2 points]** [2 points] Which aspects of this assignment did you like? Is there anything you would have changed?