UNIVERSITY OF SASKATCHEWAN

Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephone: (306) 966-4886, Facimile: (306) 966-4884

Assignment 4

Stacks and Queues, and Development Processes

Date Due: 7 February 2020, 8pm **Total Marks: 64**

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions.
- Each question indicates what to hand in. You must give your document the name we prescribe for each question, usually in the form aNqM, meaning Assignment N, Question M.
- Make sure your name and student number appear at the top of every document you hand in. These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you.
- Do not submit folders, or zip files, even if you think it will help. It might help you, but it adds an extra step for the markers.
- Programs must be written in Python 3.
- **Assignments must be submitted to Moodle.** There is a link on the course webpage that shows you how to do this.
- **Moodle will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded.
- Read the purpose of each question. Read the Evaluation section of each question.

Version History

- **30/01/2019**: released to students

Question 0 (10 points):

Purpose: To force the use of Version Control in Assignment 4

Degree of Difficulty: Easy

You are expected to practice using Version Control for Assignment 4. This is a tool that we want you to become comfortable using in the future, so we'll require you to use it in simple ways first, even if those uses don't seem very useful. Do the following steps.

1. Create a new PyCharm project for Assignment 4.
2. Use `Enable Version Control Integration...` to **initialize** Git for your project.
3. Download the Python and text files provided for you with the Assignment, and **add** them to your project.
4. Before you do any coding or start any other questions, make an initial **commit**.
5. As you work on each question, use Version Control frequently at various times when you have implemented an initial design, fixed a bug, completed a question, or want to try something different. Make your most professional attempt to use the software appropriately.
6. When you are finished your assignment, open PyCharm's Terminal in your Assignment 4 project folder, and enter the command: `git --no-pager log` (double dash before the word 'no').

Notes:

- No excuses. If you system does not have Git, or if you can't print the log as required, you will not get these marks.
- Git is installed on all Department computers (Windows and Linux). It's not a good idea to work on the same PyCharm project with Windows and with Linux. Pick one, and stick with it. Going back and forth can corrupt PyCharm's configuration files.
- If you do switch back and forth between Windows and Linux, you may have two local Git repositories, rather than just one. This is less than ideal for advanced use, but manageable for Assignment 4. Grab a git log from both systems, and submit as a single log (txt) file.
- If you are working at home on Windows, Google for how to make git available on your command-line window. You basically have to tell the Python Terminal where the git app is.

Examples of good commit messages:

- Initial commit for Assignment 4.
- Added test cases for Question 2.
- Completed Question 3.
- Fixed a bug in A4Q2: `copy()`

Examples of bad commit messages:

- lkdfjgldkfjgd
- Committing files
- Step 2
- Help! I've fallen into a Git repo and I can't get up!

What to Hand In

After completing and submitting your work for the remaining Questions, open a command-line window in your Assignment 4 project folder (or open the Terminal pane in PyCharm). Run the following command in the terminal: `git --no-pager log` (double dash before the word 'no'). Git will output the full contents of your interactions with Git in the console. Copy/paste this into a text file named `a4-git.log`.

If you are working on several different computers, you may copy/paste output from all of them, and submit them as a single file. It's not the right way to use git, but it is the way students work on assignments.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 10 marks: The log file shows that you used Git as part of your work for Assignment 4. For full marks, your log file contains
  - Meaningful commit messages.
  - At least two commits per question for a total of at least 10 commits. And frankly, if you only have 2 commits per question, you're pretending.

Question 1 (25 points):

Purpose: To practice design processes.

Degree of Difficulty: Moderate.

In this question you will apply an informal design process to the task of implementing a program at the level of Assignment 1. The purpose is to follow the process thoughtfully and reflectively. Please read the question description carefully before you start!

What to Hand In

Usually this appears last, but in this question, it's first so that you know what you have to hand in. As you can see, the code you develop is a small part of this work.

- Your plan to complete this question: `a4q1_plan.txt`
- Your design document: `a4q1_design.txt`
- Your test script: `a4q1_testing.py`
- Your Git Log: `a4q1.log`
- Your answers to the reflection questions: `a4q1_reflections.txt`
- Your Python program `a4q1.py`

You are allowed to use PDF, RTF, DOC files instead of TXT. Be sure to include your name, NSID, student number, course number and lecture section at the top of all documents.

Problem Specification

In this question you will implement a program that checks whether a $N \times N$ square of numbers is a Latin Square or not. This is similar (but not too similar) to the Magic Square Problem of Assignment 1.

A *Latin Square* is an arrangement of numbers 1 to N in a square, so that every row and column contain all the numbers from 1 to N. The order of the numbers in a row or column does not matter. Below are two squares, but only one of them is a Latin Square.

1	3	2
3	2	1
2	1	3

1	2	3	4
2	1	4	3
3	3	1	2
4	3	2	1

The square on the left is a Latin Square for the numbers 1,2,3, whose rows consist of the numbers 1,2,3 (in any order) and whose columns consist of the numbers 1,2,3 (in any order). On the right, is a 4×4 square of numbers that is *almost* a Latin Square, but the third row (and second column) is missing the value 4. There are Latin squares of size N for any positive integer N. A $N \times N$ Latin square is formally defined by the following criteria:

- Every row contains the integers 1 through N exactly once, in any order.
- Every column contains the integers 1 through N exactly once, in any order.

When the Latin square is $N \times N$ we say that it has "order N". The order tells us which numbers to look for, and how big the square is.

A Latin Square Checker is a program to check if a file contains a Latin Square or not. In detail, this program will:

- Ask for and obtain the name of a file on the console using `input()`.
- Open the named file, containing the following information for a single complete square:
  - It reads a number N on a line by itself. This will be the order of a Latin square. The order must be a positive integer, e.g., $N > 0$.
  - It reads N lines of N numbers separated by spaces, i.e., it reads console input for a square of numbers.
  - For purposes of this assignment, your program can assume that the file exists, and that the file contains complete information: there will be $N + 1$ lines as described above.
- Checks whether the sequence of numbers is a Latin square or not. Your program should display the message `yes` on the console if it satisfies the above criteria, or `no` if it does not.

The problem specification above tells you what your program should do, but not what you need to do for this question. That's next!

Task

For this question you will do the following:

- You will start by creating a plan for your work. Your plan will schedule time for the following development phases:
  - Requirements. The specification is given above, so this part of the plan includes reading the Specification section carefully.
  - Design. Your plan will indicate when you will do the design phase (day, time, location), and how long you think it will take (duration).
  - Implementation. Indicate when you will work on the implementation phase (day, time, location), using your design, and how long you think it will take (duration).
  - Testing and debugging. Indicate when you will work on testing and debugging (day, time, location), and how long you think it will take(duration).
  - Final clean up for submission. Indicate when you will work on clean-up (day, time, location), and how long you think it will take (duration).

  You will submit this plan for grading.
- You will design your program, producing a design document outlining your implementation. You will submit this design for grading.
- You will apply the iterative and incremental development process, as outlined in the lecture. Your log from using Git will be submitted showing evidence that you used this process.
- You will record the amount of time you actually spent on each phase of the plan. You will submit this information for grading.

Details

Details about the Plan On page 8 is an example table that you could use to outline your plan. It has columns for the information required above. You are not required to follow your plan exactly, to the last detail. It's a plan, not a commitment. It's helpful for monitoring your progress. You might find your plan didn't account for issues and problems that came up. That's okay. Make note of those in your reflections.

Details about the Design Your design document should describe your implementation in terms of the functions you'll need to complete the program. For each function, you must specify:

- The doc-string information as in Chapter 3: Purpose, Pre-Conditions, Post-Conditions, Return.
- Pseudo-code or informal algorithm descriptions. The form of the description is up to you.
- Black-box test cases based on your doc-string description of the function. Consider the discussion in Chapter 5 as well.

You can use the design document from Assignment 1 as an example.

Details about the Implementation, Testing, and Debugging Try to apply an incremental approach. For each function in your design:

- Implement the function according to your design.
- Add test cases from your design to the test driver script.
- Debug as necessary, adding test cases as needed (white box cases).
- When you're confident, use Git to make a commit to mark your progress, and save your work.

Keep track of your time during these phases. You'll want some objective evidence for how long things actually take, compared to what you estimated in your plan. This objective evidence will help you prepare more realistic plans in the future! How you actually complete these phases is less important than being objective about your progress.

Details about your submitted program Your program will be well documented, including doc-strings as described in Chapter 4. Any tricky parts of your algorithm will be annotated with helpful comments for future readers and markers. Your program should be correct. Try to avoid horribly inefficient algorithms. The issue of robustness concerning missing data files or corrupted data files need not be addressed in this question. The implementation goals of adaptability and reusability are not of concern for this question.

Details about your submitted Git log Question 0 also requires a git log. If you complete Question 1 before any others, the git log you submit here will be shorter than the whole git log for Question 0, but it's okay for the complete git log to include the work on this question.

Details about your reflection Your answers to the reflection questions below are graded for relevance and thoughtfulness of your answer. There are no right answers, and the only way to lose these marks is to fail to reflect meaningfully.

Reflection questions

1. How long did it take for you to create your plan?
2. Did you revise your design? In other words, did any part of your implementation turn out to be significantly different from your design? If so, briefly explain the reason for the difference(s).
3. Did any of the phases take significantly longer than you had planned? If so, explain what happened, and suggest ways to avoid this in the future.

Evaluation

- 3 marks: Your plan allocated time to every phase of the development process, and the times were plausible.
- 3 marks: Your design gave complete and useful details in terms of function descriptions, pseudo-code algorithms, and test cases.
- 3 marks. Your test script included all the test cases in your design, and possibly more.
- 3 marks: Your git log suggests that you followed an incremental development strategy.
- 3 marks: Your answers to the reflection questions were thoughtful and relevant.
- 5 marks: Your program is correct, and not terribly inefficient.
- 5 marks: Your program is well-documented, with doc-strings as described in Chapter 4, and other helpful comments as needed.

Development Plan

Phase	**Day**	**Time**	**Location**	**Duration**	*Actual Time*
Requirements					
Design					
Implementation					
Testing/debugging					
Tidying up					

Question 2 (15 points):

Purpose: To practice using a Stack as an algorithmic tool (Chapter 8).

Degree of Difficulty: Moderate.

Important Note – Read this unless you want zero marks

The purpose of this question is to practice and achieve mastery of the Stack ADT as an algorithmic tool. **Your program must use the given Stack ADT for this.**

Download the Stack ADT implementation named `TStack.py` from the Assignment 4 page on Moodle. Your script for this question should import this module.

To help you avoid errors, this implementation of the Stack ADT does not allow you to violate the ADT in a careless way. **You do not need to understand the code in the file `TStack.py`. You will not be tested on this implementation. The tricky Python code in this ADT is only used to cause a run-time error if you use the ADT the wrong way, and is not to be taken as an example of ADT design.** We will study simpler and more accessible implementations starting with Chapter 10. You should focus on using the Stack operations correctly. `TStack.py` has the same Stack ADT interface, and has been thoroughly tested. If your script does not use the Stack ADT correctly, or if you violate the ADT Principle, this implementation will cause a runtime error. If you see errors coming from the `TStack.py` module, it's almost certainly because you used the operations incorrectly.

You will get **zero marks** if any of the following are true for your code:

- Your script uses the `reverse()` method for lists, or anything similar for strings.
- Your script uses the extended slice syntax for lists or strings to reverse the data.
- Your script does not use the Stack ADT in a way that demonstrates mastery of the stack concept.

Task

Design and implement a Python script that opens a text file, reads all the lines in the file, and displays it to the console as follows:

- The lines are displayed in reverse order; the first line in the file is the last line displayed.
- The word order in the line is reversed; the first word in a line from the file is the last word displayed on a line in the console.
- The characters in each word are not reversed; the first letter of each word in the file appears as the first letter in the word displayed.

For our purposes here, a **word** is any text separated by one or more spaces, that is, exactly the strings you get when you use the string method `split()`. So normal punctuation may look a bit weird when you run this script; that's okay!

For example, suppose you have a text file named `months.txt` with the following three lines:

```
January February March April
May June July August.
September October November December
```

Running your script on the console should produce the following output:

```
December November October September
August. July June May
April March February January
```

Notice:

- The sequence of the lines displayed is reversed compared to the file.
- The sequence of words displayed on a line is reversed compared to the file.
- The sequence of characters in the words are not reversed.

What to Hand In

- Your implementation of the program: `a4q2.py`.
- Your test script: `a4q2_testing.py`
- Copy/paste a few examples of your script working on some files (this will suffice as system testing): `a4q2_output.txt`.

Be sure to include your name, NSID, student number, course number and lecture section at the top of all documents.

Evaluation

- 2 marks: Your program displays the reversed contents of the file to the console.
- 3 marks: Your program uses the Stack ADT to reverse the order of the lines in the file.
- 3 marks: Your program uses the Stack ADT to reverse the order of the words on each line of text (string).
- 4 marks: Your test script demonstrates adequate testing.
- 3 marks: Your output file demonstrates the program working on at least 3 examples.

Note: Use of `reverse()`, or extended slices, or any other technique to avoid using Stacks will result in a grade of zero for this question.

Question 3 (10 points):

Purpose: To work with a real application for stacks

Degree of Difficulty: Moderate. You have to learn a new algorithm, then implement it.

In class, we saw how to evaluate numerical expressions expressed in *post-fix* (also known as *reverse Polish notation*). The code for that is available on the course Moodle. It turns out to be fairly easy to write a similar program to evaluate expressions using normal mathematical notation.

Input

The input will be a mathematical expression in the form of a string, using at least the four arithmetic operators ($+, -, \times, /$) as well as pairs of brackets. To avoid problems that are not of interest to our work right now, we'll also use lots of space where we normally wouldn't. **We'll use spaces between operators, numbers and brackets. This space is required so that `split()` can be used to split the different tokens.** Here's a list of expressions for example:

```
example1 = '( 1 + 1 )'                # should evaluate to 2
example2 = '( ( 11 + 12 ) * 13 )'   # should evaluate to 299
```

Notice particularly that we are using brackets explicitly for every operator. In every-day math, brackets are sometimes left out, but in this is not an option here. **Every operation must be bracketed!** The brackets and the spacing eliminate programming problems that are not of interest to us right now.

Hint: You will find it useful to split the string into sub-strings using `split()`, and even to put all the substrings into a Queue, as we did for the PostFix program.

Algorithm

We will use two stacks: one for numeric values, as in the PostFix program, and a second stack just for the operators. We take each symbol from the input one at a time, and then decide what to do with it:

- If the symbol is `'('`, ignore it.
- If the symbol is a string that represents a numeric value (use the function `isfloat()`, provided in the script `isfloat.py`), convert to a floating point value and push it on the numeric value stack.
- If the symbol is an operator, push the operator on the operator stack.
- If the symbol is `')'`, pop the operator stack, and two values from the numeric value stack, do the appropriate computation, and push the result back on the numeric value stack.

You should write a function to do all this processing.

Since the objective here is to practice using stacks, you must use the Stack ADT provided for this assignment. Also, you may assume that the input will be syntactically correct, for full marks. For this question your program does not need to be robust against syntactically incorrect expressions.

Using the Stack and Queue ADTs

Download the ADT implementations named `TStack.py` `TQueue.py` from the Assignment 4 page on Moodle. To help you avoid errors, these implementations do not allow you to violate the ADT in a careless way. **You do not need to understand the code in the files. You do not even need to look at it.** We will study simpler and more accessible implementations later in the course. You should focus on using the ADT operations correctly. These versions have the same ADT operations, and have been thoroughly tested. If your script does not use these ADTs correctly, or if you violate the ADT Principle, a runtime error will be caused. If a runtime error points you to the code in the ADT, the error is almost certainly in your script, not the ADT code.

Testing

This is the first script whose testing needs to be very diligent, and will end up being extensive. Write a test script that checks each arithmetic operation in very simple cases (one operator each), and then a number of more complicated cases using more complicated expressions. Your test script should do unit testing of your function to evaluate expressions. You can add tests of any other functions you write, but focus on testing the expressions.

What to Hand In

- Your function, written in Python, in a file named `a4q3.py`
- Your test-script for the function, in a file called `a4q3_test.py`

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 6 marks: Your evaluation function correctly evaluates expressions of the form given above. It uses two Stacks, both are created by the TStack ADT.
  - 1 mark: The evaluation function correctly handles numeric data by pushing onto a numbers stack.
  - 1 mark. The evaluation function correctly handles operators by pushing on an operator stack.
  - 3 marks. The evaluation function correctly evaluates expressions when a `')'` is encountered. Two values are popped from the numbers stack, and an operator is popped from the operator stack. The correct operation is performed, and the result is pushed back on the numbers stack.
  - 1 mark. When there is nothing left to evaluate, the result is popped from the numbers stack.
- 4 marks: Your test script covers all the operations individually once, and in combination.
  - 2 marks: Every operator is tested.
  - 2 marks: Testing includes at least one expression with several nested operations.

Question 4 (4 points):

Purpose: To introduce students to the concept of a REPL. Complete this if you have time.

Degree of Difficulty: Easy if Question 3 is working correctly.

The term "REPL" is an acronym for "read-eval-print loop". It is the basis for many software tools, for example, the UNIX command-line (PyCharm Terminal) is essentially a REPL, and the Python interactive environment is a very sophisticated REPL, and literally hundreds of other useful applications: R, MATLAB, Mathematica, Maple, to name a few.

A REPL is basically the following loop:

```
while True:
    prompt for and Read a command string from the console
    Evaluate the command string
    Print the resulting value to the console
```

In this question, you'll implement a REPL for your evaluation function from Question 3. You could make it slightly more sophisticated by allowing the user to type something like "quit" which will avoid evaluation and terminate the loop. Implement your REPL in a separate script, and import your evaluation function as a module.

An example run for your script might be as follows:

```
Welcome to Calculator.  Let's calculate!
> ( 3 + 14 )
17.0
> ( ( 11 + 4 ) / 6 )
2.5
> quit
Thanks for using Calculator!
```

This is a script you can run from PyCharm, or on the command line. The first line and last line are not part of the loop, and are just present to create a friendly context. The `'>'` are the prompts displayed by the REPL. The expressions (e.g., `'( 3 + 4 )'`) are typed by the user. The results (e.g. `7.0`) are displayed by the REPL.

What to Hand In

- Your REPL, written in Python, in a file named `a4q4.py`
- A file named `a4q4-output.txt` containing a demonstration of your REPL working, using copy/paste from the PyCharm console.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 2 marks: Your script `a4q4.py` contains a REPL that uses `a4q3.py` to evaluate simple arithmetic expressions.
- 2 marks: You demonstrated your REPL in action. It doesn't have to be cool.