UNIVERSITY OF SASKATCHEWAN

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

Assignment 4

Stacks and Queues

Date Due: June 13, 2019, 7pm **Total Marks: 58**

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

- **06/07/2019**: 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. 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').
   Note: You might have trouble with this if you are using Windows. Hopefully you are using the department's network filesystem to store your files. If so, you can log into a non-Windows computer (Linux or Mac) and do this. Just open a command-line, `cd` to your A4 folder, and run `git --no-pager log` there. If you did all your work in this folder, git will be able to see it even if you did your work on Windows. Git's information is out of sight, but in your folder.
   Note: 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.

You may need to work in the lab for this; Git is installed there.

What to Hand In

After completing and submitting your work for Questions 1-5, open a command-line window in your Assignment 4 project folder. 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` or `a4-git.text`.

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 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 10 commits, you're pretending.

Question 1 (14 points):

Purpose: Primarily, to practice using a Stack as an algorithmic tool (Chapter 8). Also, to practice designing and developing a short Python script (Chapter 7), to develop your testing skills (Chapter 5), and to practice using Version Control (Lab 2), just to get into the habit.

Degree of Difficulty: Moderate. The difficulty is in trying all these concepts at the same time!

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. When you use any function besides the ones in an ADT to work with its data structure, you are violating the ADT principle. For example, you would violate the Stack ADT when adding a new element if you use the append() function from the list ADT instead of the push() from the stack ADT. **You do not need to understand the code in the file `TStack.py`. You will not be tested on this implementation.** 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 words in the line are not reversed; the first word in a line from the file is the first word displayed on a line.
- The characters in each word are reversed; the first letter of each word in the file appears as the last 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:

```
rebmetpeS rebotcO rebmevoN rebmeceD
yaM enuJ yluJ .tsuguA
yraunaJ yraurbeF hcraM lirpA
```

Notice:

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

In Question 2, the task will be to get your program to run in the Terminal, using command-line parameters as in Lab 4. For this question, it's fine if your script runs by asking for a filename, or simply has the filename hard-coded in a variable assignment in your script.

Testing

You should test your functions individually (unit testing), and then working together (integration testing). If your functions read files, they will be harder to test. If your functions produce console output, they will be harder to test. However, if your functions have arguments that are lists, or strings, and if they return lists or strings, they will be easier to test. This is a design decision! Your system testing can be simple: just show your script working on a few files. You can create some of your own.

Design

Practice your development and design skills! This script is small enough that you don't actually need them, but that means it's small enough to see how you could come up with a development plan. Apply the iterative and incremental model as discussed in Chapter 7. Don't just think about the end product, but also your testing. Design functions that will work together to get your job done, but can also be tested easily. Before you sit down to implement your design, come up with test cases. Try to make an incremental development plan so that at every step, you can make a git commit for a program that works, and has been tested.

What to Hand In

- Your implementation of the program: `a4q1.py`.
- Your test script, including unit tests only: `a4q1_unit.py`
- Your test script, including integration tests only: `a4q1_integration.py`
- Copy/paste a few examples of your script working on some files (this will suffice as system testing): `a4q1_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 characters in each word on a line of text (string).

- 4 marks: Your test script demonstrates adequate testing.
- 2 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 2 (20 points):

Purpose: To work with a real application for stacks and be more familiar to work with an ADT that was not developed by you

Degree of Difficulty: Moderate

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. 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 in 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 ADT

Download the Stack ADT implementation named `TStack.py` from the Assignment 4 page on Moodle. 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 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 Stack ADT 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, a runtime error will be caused.

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 `a4q2.py`
- Your test-script for the function, in a file called `a4q2_test.py`

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

Evaluation

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

Question 3 (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 `a4q3.py`
- A file named `a4q3-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 `a4q3.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.

Question 4 (10 points):

Purpose: To work with ADTs on the ADT programming side.

Degree of Difficulty: Moderate. The implementation is not difficult, but there may be some initial confusion about the difference between the Queue ADT as presented in class, and the one you are working on here.

Set-up

The Queue ADT as we have studied allows storage of any number of data values: the Queue is always big enough to store any amount of data. While this is useful for most cases, it's not always realistic. If we're programming very small devices like a smart watch, or a microprocessor controlling a refrigerator, memory is not going to be as generous as with a notebook or desktop computer. Furthermore, we can imagine simulations (like the M/M/1 simulation in Chapter 9) where an infinite queue is really unrealistic. Some coffee shops operate in the real world, and fire regulations prohibit infinitely many customers waiting in line!

Task

In this question, we'll implement a variant of the Queue ADT with a given maximum capacity (we'll call it a *finite capacity queue*, or FCQueue). With 2 exceptions, the operations on FCQueue have the same effect as the operations on a normal Queue. There are 2 exceptions:

1. The create operation takes an argument, an integer which defines the capacity of the FCQueue. Once created, an FCQueue cannot store more items than the capacity.
2. The enqueue operation will work normally so long as the number of values stored is less than the capacity. Any attempt to enqueue a value to a full queue will result in the new value being dropped, and not stored in the queue at all. In the real world, this is like a customer walking away from the coffee shop if the queue is too long.

Start by downloading the FCQueue ADT module `FCQueue.py`. It's a partial implementation of all the queue operations; there are function definitions, with precise interface documentation, but trivial behaviour. Your task is to complete this ADT so that it behaves as if it had a finite capacity. The ADT has the following operations:

- The `create()` operation returns a new empty queue with a given capacity `cap`. The code is given to you:

```
1  def create(cap):
2      """
3      Purpose
4          creates an empty queue, with a given capacity
5      Return
6          an empty queue
7      """
8      b = dict()
9      b['storage'] = list()  # data goes here
10     b['capacity'] = cap    # remember the capacity
11     return b
```

  Notice that the value returned is a record (dictionary), with 2 key-value pairs. The data should be stored in the list associated with `'storage'`. Don't change the capacity at all!

- The `is_empty()` operation returns True if the queue is empty. You'll have to look at the storage to complete this operation.

- The `size()` operation returns the number of elements stored in the queue. You'll have to look at the storage to complete this operation.
- The `enqueue()` operation adds a new data element to the back of the queue. However, if the number of values already in the queue is equal to the capacity given to the create operation (above), the value should not be added; only add the value if the capacity would not be exceeded. You'll have to look at the capacity and the storage to complete this operation.
- The `dequeue()` operation removes a data element from the front of the queue. You'll have to look at the storage to complete this operation.

  Clarification: This operation should be designed so that, if called when the FCQueue is empty, a run-time error is caused. This is not something that requires additional programming! That is, no need to have an if statement for when your Queue is empty. You'll notice that a runtime error is caused when dequeue is called on an empty Queue, and it's just the result of asking for a value from an empty list. That's good enough for us for now.
- The `peek()` operation returns a reference to the data element at the front of the queue. You'll have to look at the storage to complete this operation.

  Clarification: This operation should be designed so that, if called when the FCQueue is empty, a run-time error is caused. See the above clarification.

To implement these operations, you can refer to the implementation of the Queue ADT given in the readings, Chapter 10. But the value returned by the `create()` operation is a record (i.e., a dictionary), not a simple list, so you'll have to adapt and merge the ideas together. Understanding this is the only difficult part of this question, and once you get it, the rest will fall into place easily.

Testing

We have provided a test script, which you can run to check your implementation. It will import `FCQueue.py`, and run a bunch of unit and integration tests. The partial implementation will cause a number of failed tests, and a run-time error. You task is to implement the operations so that every test passes. You are encouraged to browse the test cases, and see how they were implemented, as you'll be doing similar testing in the future. You are not required to add more test cases to the script, but you are permitted to do so, especially if you're debugging.

The markers will apply a similar set of tests, and your grade will depend on how many tests pass.

What to Hand In

Hand in your implementation of `FCQueue.py`. Don't rename it. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 10 marks: Your implementation passes all tests.