Lab 8: Stacks and Queues

CSE/IT 107

Introduction

The purpose of this lab is to dig a little deeper into stacks and queues.

Coding Conventions

Follow PEP-8 recommendations for your code.

Problems

Make sure your source code files are appropriately named. Make sure your code has a main function; use `boilerplate.py` you created in Lab 1.

As you saw last week, one nice feature of lists in Python is that you have built-in methods for stack operations: push and pop. Pushing (adding an element) is implemented via `append()` and popping (remove the top element of the stack) an element is accomplished via `pop()`. Besides evaluating postfix and prefix expressions, stacks have a number of other uses.

1. Write a program that converts a decimal number to a hexadecimal number. Name your source code file `dec2hex.py`. The file `dec2bin.py` is provided (`lab8.tar.gz`) to get you started. Make sure your program accepts a command line argument for the decimal number.

2. Write a program that uses a stack to reverse lines of a file. Name your source code file `reverse_file.py`.

   To do this with a stack, open a file, read in the lines and push them on a stack. Open a new file, and pop the lines into the new file. You should strip off the new line as you are processing lines. Your output should be a new file that is the reverse of the original file. Pass in filenames on the command line. If only one file is given the original and the new file are the same and the writing of the file is done in place destroying the original file. If two command line arguments are given the first one is the original file and the second one is the new file to create. Use a try-except block to capture the error if the file you want to open doesn't exist. Reverse the files `words.txt` and `alice.txt`, both of which are on Moodle in `lab8.tar.gz`. Running the command on a reversed file should, of course, produce the original file.

3. Building on the previous problem, write a program that not only reverse the lines in a file, but reverse the words in the line as well. Name your source code file `reverse_file_words.py`.

   Assume words are separated by spaces. Test with `alice.txt`. Running the command on the reversed file should, of course, produce the original file.

4. Write a program that checks for balanced "parenthesis" in Python programs. Name your source code `balance.py`

   "Parenthesis" in Python are left and right parenthesis, single and double quotes, left and right braces and brackets. "Parenthesis" can also be nested. For example, `a = [[1]]` is valid Python. Accept a name of a Python program as a command line argument. Use a try-except block to test to make sure the file exists. An algorithm to check for balanced "parenthesis" is as follows:

   (a) Start with an empty stack
   (b) Scan the expression from left $\rightarrow$ right
   (c) If you read a left "parentheses" (or brace, braces, etc.) push it onto the stack
   (d) When you encounter a right "parentheses" check that the top of the stack contains a matching left "parenthesis". If so, pop the top item off the stack. An error occurs if right "parenthesis" occurs and there is not a matching left "parenthesis".
   (e) When finished reading the expression and the stack is empty, the expression contains balanced "parenthesis". If something is left on the stack, something is wrong.

   You are checking to see if the whole file is balanced, not just a particular expression. Read the file in line-by-line and check each line. "Parenthesis" may extend across multiple lines.

   If the file is balanced print out the phrase "File <filename> is balanced", where you replace <filename> with the name of the file you tested. If the file is unbalanced, exit the program and report the line and the type of syntax (brace, parenthesis, etc.) that produces an unbalanced "parentheses".

5. Write a program that implements the selection sort. Name your source code `selection_sort.py`

   You will implement a selection sort using two stacks. The original stack holds items in two sections. Items near the bottom of the stack are sorted and those near the top are not. Initially, no items are sorted, and all items are in the not sorted section of the stack. The second stack is a temporary stack.

   For each element in the original stack, move all the unsorted items to the temporary stack, keeping track of the largest item in a variable. After all unsorted items have moved to the temporary stack, push the largest item to the original stack and move all items, except the largest value, from the temporary stack back to the original stack. Repeat until all items in the original stack are sorted.

At each step of the way there are N, N - 1, N - 2, ..., 2, 1 unsorted items in the original stack that need to be moved to the temporary stack.

6. Write a program that finds all palindromes in the file `words.txt`. Name your source code `palindrome.py`.

   Remember, a palindrome is a string that reads the same forward as well as backwards. For example, racecar, noon, radar, etc.

   You will use a stack to find palindromes. To find a palindrome with a stack, you do the following:

   (a) Push all letters of a word up to the middle of the word onto a stack. Account for even and odd word lengths.
   (b) For the second half of the word, compare each letter with the top of the stack. If they are the same pop the stack and go to the next element. If they are not the same, the word is not a palindrome.
   (c) Continue until the stack is empty (a palindrome) or the string is not a palindrome.

   Assume palindromes have a length of at least two characters. The single characters a, b, c, ..., z in the file `words.txt` indicate sections. For each word that has a length of at least 2, test if it is a palindrome. If it is is a palindrome, print the word to a file called `palindromes.txt`, one line per palindrome.

   Determine the percentage of words that are palindromes in the file. Write that to the end of `palindromes.txt`

Queues

It is not very efficient to use a list to implement a queue – a list where you insert at one end of the list and delete at the other end. Another name for a queue is a first in first out (FIFO) list. Think of standing in a checkout line. Lists are not efficient implementation of queues in python because both list methods, insert and remove, move elements to maintain the list. For efficiency reasons, queues are implemented using `collections.deque`, which you can import using the line: `from collections import deque`. Two methods deque implements are `deque.append()` which adds an element to the end of the deque and `deque.popleft()` which removes the first element in the deque in an efficient manner.

7. Write a program that uses a queue to reverse the elements in a stack. Name your source code `reverse_stack.py`. Push the sequence 1, 2, 3, ..., 50 onto a stack (50 should be the top element when you are done pushing). Now reverse the stack using a queue. The top element should be 1 when you are finished pushing the elements back on the stack. Print the original stack and the reversed stack to the terminal.

8. Write a program that implements a radix sort. Name your source code `radix.py`.

   A radix sort for base 10 integers is a based on sorting punch cards, but it turns out the sort is very efficient. The sort utilizes a main bin and 10 digit bins. Each bin acts like a queue and maintains its values in the order they arrive. The algorithm begins by placing each number in the main bin. Then it considers the ones digit for each value. The first value is removed and placed in the digit bin corresponding to the ones digit. For example, 534 is placed in digit bin 4 and 662 is placed in the digit bin 2. Once all the values in the main bin are placed in the corresponding digit bin for ones, the values are collected from bin 0 to bin 9 (in that order) and placed back in the main bin. The process continues with the tens digit, the hundreds, and so on. After the last digit is processed, the main bin contains the values in order.

   Use randint, found in random, to create random integers from 1 to 100000. Use a list comphrension to create a list of varying sizes (10, 100, 1000, 10000, etc.). To use indexing to access the digits first convert the integer to a string. For this sort to work, all numbers must have the same number of digits. To zero pad integers with leading zeros, use the string method `str.zfill()`. Once main bin is sorted, convert the strings back to integers.

Submission

Create a tarball of your *.py files.

```
tar czvf cse107_firstname_lastname_lab8.tar.gz *.py
```

To check the contents of your tarball, run the following command:

```
tar tf cse107_firstname_lastname_lab8.tar.gz *.py
```

You should see a list of your Python source code files.

Upload your tarball in Moodle before the start of you next lab.