Assignment 7

Design, Testing, Defensive Programming, and Style

Date Due: March 26, 2017, 7pm **Total Marks: 45**

Question 1 (15 points):

Purpose: To practice design and good style in a relatively simple application.

Degree of Difficulty: Moderate, because of the time you should be spending in preparation.

Write a Python program that reads a named text-file from the current working directory (e.g., a file with `.txt` or `.py` extension) and displays to the console the 10 most frequently used string characters (letter, number, or symbol) along with the number of times it appears in the file. If the file has fewer than 10 unique characters, your program should display the frequencies of all of them.

Your program should run from the command-line, and use a file-name that was given on the command-line. For example, suppose the file `hello.txt` contains the following two lines of text:

```
Hello,
world!
```

Running your program on the command line would produce the following results:

```
UNIX$ python3 a7q1.py hello.txt
'l' : 3
'o' : 2
',' : 1
'w' : 1
'e' : 1
'H' : 1
'd' : 1
' ' : 1
'!' : 1
'r' : 1
UNIX$
```

When characters have the same frequency (i.e., ties), the order is not important.

Since this is an application, your testing for this program has to be in an external script.

Hints:

- You probably have some ideas about how you're going to do this. Take the opportunity to practice breaking the problem into functional units, designing interfaces to functions, thinking about test cases. The actual program you write is not important. The process is important.
- Use a dictionary.
- Use Python's list method, sort. It has some optional keyword parameters that will help you. Part of this assignment is to read official Python documentation, and figure out how to use Python tools that you may not have learned yet.
- Test your program on various files. Specifically, make sure you try files that have multiple lines. Also test your program on empty files, and some larger files.

What to Hand In

Hand in a well-written, suitably documented program named `a7q1.py`. Pay attention to your programming style, and use design, testing, and defensive programming to produce the best application you can produce in the time available. You do not need to hand in any testing. That does not mean you should do no testing. The markers will run your program on test data. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your program produces the correct results on the marker's test files.
- 10 marks: Your program is well-written, using good style, and shows evidence of good design.

Solution: A model solution appears in the file `a7q1.py`. In the model solution, the core of the solution lies in two functions. The first, `count_characters()` creates a dictionary:

```
def count_characters(contents):
    """
    Purpose:
        Count the characters in the contents.
    Pre-conditions:
        :param contents: a list of strings.
    Return:
        :return: a dictionary with the characters and frequencies
    """
    freqs = dict()
    for line in contents:
        for c in line:
            if c in freqs:
                freqs[c] += 1
            else:
                freqs[c] = 1
    return freqs
```

There is more than one way to do this, including an initialization of the dictionary with zero counts, or the use of a default value with the `get()` method. Scanning the list once like this is fast, requiring $O(n)$ dictionary accesses.

There are worse ways to do this. For example, you could scan the whole string once for each character, which could take as much as $O(n^2)$ time, if done particularly poorly.

The other function in the model solution is `display_top()`, which sorts the list and pulls a slice of 10 elements:

```
def display_top(frequencies, k):
    """
    Purpose:
        Display the top k most frequently occuring keys.
    :param frequencies: a dictionary of keys and counts.
    :param k: an integer, k >= 0
    :return:
    """
    def comp(p):
        return p[1]

    assert k >= 0, 'Must have non-negative ordinal'

    pairs = [(c, frequencies[c]) for c in frequencies]
    pairs.sort(key=comp, reverse=True)
    for c, v in pairs[:k]:
        print("'" + c + "'", v)
```

To sort the list, the model solution tells sort to use the function `comp` to pull the frequency from a character-frequency pair.

Again, there are a lot of acceptable ways to do this. It is quite appropriate to call the sort method with various values for `key`, such as `getitem`, or other method. Finding some technique to do this task was part of the exercise, so any program that is not outrageously inefficient is probably okay.

The model solution goes to great lengths to document the program, and demonstrate good design. It uses a few assertions, to ensure that error situations are caught.

An acceptable solution to this task could easily have fewer functions. The functional decomposition is part of the exercise.

Notes for the Markers:

- Finding some technique to do this task was part of the exercise, so any program that is not outrageously inefficient is probably okay.
- The functional decomposition is part of the exercise. If the program is well organized, and documented, then almost any collection of functions is okay.
- Deduct marks for disorganized code, or code that has little to no documentation.
- Deduct marks for code that works, but does not live up to the design philosophy of keeping things simple and easy to understand.
- As a guideline, the marks for the correctness are allocated as follows:
  - If the code is obviously fine, 5/5.
  - If it's not clear, run the code on the two given marking examples.
  - If it works on both examples, 5/5.
  - If it works on one but not the other, 3/5
  - If it doesn't work on either example 1/5
- As a guideline, the marks for the design are allocated as follows:
  - 10/10 for well organized well-documented code. It doesn't have to me as nice as the model solution to get 10/10.
  - 8/10 for code that shows an effort was made, but the effort could be improved.
  - 5/10 for a program that has no documentation, but is otherwise fine.
  - 3/10 for a program that is a complete mess.

  (Yes, that means the minimum grade is 3/10 here, as long as someone handed something in).

Question 2 (30 points):

Purpose: To practice design and good style when the program is not as easy. Break this problem in to pieces, and solve them one at a time, not all at once.

Degree of Difficulty: Tricky

Write a program to decode messages! Your program will read from a file, containing information about the code and the encoded message, and your program will decode it, and display the decoded message to the console. For example, a small example file is the following:

```
10
1110:' '
1100:'!'
1010:','
1011:'H'
1101:'d'
1111:'e'
01:'l'
100:'o'
000:'r'
001:'w'
1011111101011001010111000110000001110111100
```

The first line of the file is the number of unique characters in the message. In this example, there are 10 characters. The next 10 lines of the file are the code-lines. In the example, there are 10 code-lines, where the code (a sequence of `0` and `1` is followed by a character. Observe that the code is separated from its character by a colon `:`, and that the character is delimited by the quote `'` character. The last line of the file is the message.

Here's the coded message and the decoded result, aligned in a way to make it easy to see:

```
1011 1111 01 01 100 1010 1110 001 100 000 01 1101 1100
  H    e   l  l  o    ,        w   o   r  l   d    !
```

Note very carefully, because it is important, that the codes have different lengths. For example, the code for `'l'` has length 2, whereas the code for `'H'` has length 4. Also note that the code for `'l'` is `01`, and `01` never starts the code for any other character. Go look! **In general, the complete code for a character never appears as the initial sequence for any other character.**

This observation is necessary for decoding. To decode the message, we pull digits one by one from the front of the coded message, and check whether the digits we've pulled forms one of the codes. If so, we put the code's character in the output string. If not, we take another digit from the coded message. The decoding process is to build up potential codes from the message one digit at a time, until we find an actual code, and then we start over again with one digit from the coded message.

A short example of the decoding process, starting from the front:

- We start with the first digit in the coded message, `1`. Observe that there is no character whose complete code is `1`.
- Then we take another digit giving us `10`, but `10` is not a complete code for any character.
- We take another digit, giving us `101`, but it's still not a complete code (though there are 2 characters whose code starts with `101`).
- When we take the 4th digit, we get `1011`, there is no ambiguity: the only possibility is that `1011` is the code for character `H`. There is no other character whose code starts with the code for `H`. We put `H` in our output string.

- We restart the decoding at digit 5, with `1`.

More complicated examples will have much more interesting codes, with far more than 4 digits per character. You will find a set of examples on Moodle.

Running your program on the command line would produce the following results:

```
UNIX$ python3 a7q2.py example1.txt
abcdefg
UNIX$ python3 a7q2.py example2.txt
Hello, world!
```

Your task is to read the file, and decode the message, using the codes provided in the file. A collection of example files is posted to Moodle.

Hints:

- Read all the lines in the file.
- Split each code line using the string method `split(':')`, since the `':'` separates the code from the character. You may assume that the message will not contain the `':'` (colon) character.
- Create a dictionary with the code as the key, and the character as the value.
- Use the dictionary to decode the message. Check if your potential code is in the dictionary using `if code in codes`.

What to Hand In

Hand in a well-written, suitably documented program named `a7q2.py`. Pay attention to your programming style, and use design, testing, and defensive programming to produce the best application you can produce in the time available. You do not need to hand in any testing. That does not mean you should do no testing. The markers will run your program on test data. Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 10 marks: Your program produces the correct results on the marker's tests.
- 20 marks: Your program is well-written, using good style, and shows evidence of good design.

Solution: A model solution is given in the file `Decoder.py` (which was distributed as part of Assignment 8 Question 3).

In the model solution, the core of the solution lies in two functions. The first, `build_decoder()` creates a dictionary:

```
    Purpose:
        Build the dictionary for decoding from the given list of code-lines
    Preconditions:
        :param code_lines: A list of strings of the form "CODE:'<char>'"
    Return:
        :return: a dictionary whose keys are 'CODE' and values are <char>
    """
    codec = {}
    for code_str in code_lines:
        cchr = code_str.split(':')
        code = cchr[0]
        char = cchr[1]
        codec[code] = char[1]   # The input file has ' ' around the character
    return codec
```

The model solution makes the process as clear as possible. This function also highlights a fairly serious design flaw: the use of the colon character `':'` to separate code-character in the file. Using a character to separate these two items makes splitting the line easy, but such an easy split will cause havoc if the colon character appears as one of the encoded characters.

The second function is `decode_message()` which does the hard work.

```
    Purpose:
        Decode the message using the decoder.
    Preconditions:
        :param coded_message: An encoded string consisting of digits '0' and '1'
        :param codec: a Dictionary of coded_message-character pairs
    Return:
        :return: A string decoded from coded_message
    """
    decoded_chars = []
    first = 0
    last = first + 1
    while first < len(coded_message):
        partial_code = coded_message[first:last]
        if partial_code in codec:
            decoded_chars.append(codec[partial_code])
            first = last
            last = first + 1
        else:
            last += 1
    return ''.join(decoded_chars)
```

The model solution uses two variables to keep track of indices:`first` keeps track of the start of a code, and `last` keeps track of the last character in a code. The variable `last` keeps advancing by 1, until a slice of the coded message is actually a code in the dictionary.

There are many ways to achieve the decoding. Consuming bits from the encoded message, and constructing a string with the characters is fine. The main thing is to keep trying codes until one is found to be in the dictionary.

Notes for the Markers:

- Finding some technique to do this task was part of the exercise, so any program that is not outrageously inefficient is probably okay.
- The functional decomposition is part of the exercise. If the program is well organized, and documented, then almost any collection of functions is okay.
- Deduct marks for disorganized code, or code that has little to no documentation.
- Deduct marks for code that works, but does not live up to the design philosophy of keeping things simple and easy to understand.
- As a guideline, the marks for the correctness are allocated as follows:

  - If the code is obviously fine, 10/10.
  - If it's not clear, run the code on the two given marking examples.
  - If it works on both examples, 10/10.
  - If it works on one but not the other, 6/10
  - If it doesn't work on either example 2/10

- As a guideline, the marks for the design are allocated as follows:
  - 20/20 for well organized well-documented code. It doesn't have to me as nice as the model solution to get 20/20.
  - 16/20 for code that shows an effort was made, but the effort could be improved.
  - 10/20 for a program that has no documentation, but is otherwise fine.
  - 6/20 for a program that is a complete mess.