Coding Assignment Writeup [Template]

Posting ID: 5582-527

1. Reflection

The objective of the assignment is to arrange a given paragraph in such a way that the characters in each line do not exceed a maximum length (M) and the sum of cubes of empty spaces left at the end of each line (except for the last line) is minimum. The expected output is the overall cost and the final arrangement of the paragraph. The input and verification logic is already provided in the scaffolding code and we only need to provide the logic to evaluate cost and final arrangement of paragraph.
This is achieved through dynamic programming as follows:

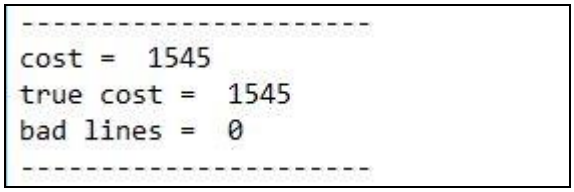
- We first evaluate the empty spaces left at the end of each line. These lines are formed by all possible combinations of words starting from first word in the paragraph to the last one. We use *i, j* to denote these sets (*i* <= *j*).
- We then evaluate the cubes of these values and make the optimum choice of cost for each line. Once again, these lines are all possible combinations of words starting from first word in the paragraph to the last one.
- Using this choice of cost for each line, we determine the final cost (which is the last element in the list in our case) and also the final arrangement of the paragraph.

2. Testing Output

Below 2 screenshots are taken from the "output.log" file (to which the output returned from the implementation is written) that indicate the output obtained from the 2 input text files given with the scaffolding code:

- kubla_kahn.txt:
    - The evaluated optimum cost is in the first line. The expected optimum cost is in the second line. As we can see that they both match.
    - Last line shows the number of lines that might have been arranged incorrectly. This value is zero in this implementation.

    So, based on the above 2 points, we can say that the output generated by this implementation is correct for this text file.



- magna_carta.txt:
    - The evaluated optimum cost is in the first line. The expected optimum cost is in the second line. As we can see that they both match.
    - Last line shows the number of lines that might have been arranged incorrectly. This value is zero in this implementation.

    So, based on the above 2 points, we can say that the output generated by this implementation is correct for this text file.

```
----------------------
cost =  13910
true cost =  13910
bad lines =  0
----------------------
```

3. Static Analysis / Compilation Output

Code compilation and execution is without any issues as seen in the below screenshot.

```
C:\Users\MadhulikaBushi\Anaconda\python.exe "C:/Users/MadhulikaBushi/Desktop/501/Coding Assignments/Coding Assignment 4/print_neatly_test.py"
Running....
        1795 function calls in 0.057 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.002    0.002    0.057    0.057 <string>:1(<module>)
        1    0.052    0.052    0.055    0.055 ch15.py:11(print_neatly)
      353    0.000    0.000    0.000    0.000 {len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1439    0.003    0.000    0.003    0.000 {range}


        24531 function calls in 10.284 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.380    0.380   10.284   10.284 <string>:1(<module>)
        1    9.371    9.371    9.904    9.904 ch15.py:11(print_neatly)
     4838    0.000    0.000    0.000    0.000 {len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
    19690    0.532    0.000    0.532    0.000 {range}


Done

Process finished with exit code 0
```

Flake8 Output:

```
C:\Users\MadhulikaBushi\Desktop\501\Coding Assignments\Coding Assignment 4>flake
8 --max-complexity 9 ch15.py

C:\Users\MadhulikaBushi\Desktop\501\Coding Assignments\Coding Assignment 4>
```

Doctest Output:

```
C:\Users\MadhulikaBushi\Desktop\501\Coding Assignments\Coding Assignment 4>pytho
n -m doctest -v ch15.py
Trying:
    print_neatly(["World","Map"], 10)
Expecting:
    (0, 'World Map')
ok
2 items had no tests:
    ch15
    ch15.get_cost_and_paragraph
1 items passed all tests:
   1 tests in ch15.print_neatly
1 tests in 3 items.
1 passed and 0 failed.
Test passed.
```

4. Source Code

```
1.  """
2.  This file provides implementation of Print Neatly functionality.
3.  It basically takes a list of words and max_line length (M) as inputs and
4.  provides the most optimum way of arranging the words in a paragraph of
5.  max_line length = M. It also provides the cost associated with this arrangement.
6.  """
7.  import sys
8.  INFINITY = sys.maxint
9.
10.
11. def print_neatly(words, M):
12.     """
13.    Print text neatly.
14.    Parameters
15.    ----------
16.    words: list of str
17.        Each string in the list is a word from the file.
18.    M: int
19.        The max number of characters per line including spaces
20.
21.    Returns
22.    -------
23.    cost: number
24.        The optimal value as described in the textbook.
```

```
    text: str
        The entire text as one string with newline characters.
        It should not end with a blank line.

    Details
    -------
        Look at print_neatly_test for some code to test the solution.
    >>> print_neatly(["World","Map"], 10)
    (0, 'World Map')
    """
     word_count = len(words)
     word_length = [0] * word_count
     extra_spaces = [[0 for x in range(word_count)] for x in range(word_count)]
     line_cost = [[INFINITY for x in range(word_count)] for x in range(word_count)]

     optimum_cost_word_position = [0] * word_count

     # finding the number of characters in each word
     for i in range(0, word_count):
         word_length[i] = len(words[i])

     # finding the extra spaces and cost associated with all possible combinations of words per
    line.
     for i in range(0, word_count):              # let i be the first word of a given line
         for j in range(i, word_count):  # let j be the last word of a given line. i <= j
             if i == j:
                 extra_spaces[i][j] = M - word_length[i]
             else:
                 extra_spaces[i][j] = extra_spaces[i][j-1] - word_length[j] - 1

             if extra_spaces[i][j] >= 0:         # combination of words will fit in a line.
                 if j == word_count-1:           # last line
                     line_cost[i][j] = 0
                 else:
                     line_cost[i][j] = (extra_spaces[i][j])**3

     cost, optimum_cost_word_position = get_cost_and_paragraph(word_count, line_cost, optimum_co
    st_word_position)

     # preparing the text output
     last_word_pos = word_count-1
```

```
64.     first_word_pos = optimum_cost_word_position[last_word_pos]
65.     text = ''
66.     while first_word_pos >= 0 and last_word_pos >= 0:
67.         temp_text = ''
68.         for i in range(first_word_pos, last_word_pos+1):
69.             temp_text += ' ' + words[i]
70.         last_word_pos = first_word_pos - 1
71.         first_word_pos = optimum_cost_word_position[last_word_pos]
72.         text = '\n' + temp_text[1:] + text
73.
74.     return cost, text[1:]
75.
76.
77. def get_cost_and_paragraph(word_count, line_cost, optimum_cost_word_position):
78.     """
79.    finding best combination of words for each line and the cost assoicated with it.
80.    :return: cost and best combination of words for each line
81.    """
82.     cost = [INFINITY] * word_count
83.     for j in range(0, word_count):              # let j be the last word of a given line
84.         for i in range(0, j+1):                 # let i be the first word of a given line. i <=
    j
85.             if i == 0:
86.                 if line_cost[i][j] < cost[j]:                   # updating the cost when
    considering words from 1 to j.
87.                     cost[j] = line_cost[i][j]                   # updating position when new
    "least cost" is identified.
88.                     optimum_cost_word_position[j] = i
89.             # check if sum of cost of words from [i to j] & cost at i-1 is less than cost at j
90.             elif cost[i-1] + line_cost[i][j] < cost[j]:
91.                 cost[j] = cost[i-1] + line_cost[i][j]  # updating the cost when considering
    words from 1 to j.
92.                 optimum_cost_word_position[j] = i          # updating position when new "least
    cost" is identified.
93.     return cost[-1], optimum_cost_word_position
```

Revised rubric for coding assignments.

This is a 5-point rubric for coding projects. Graders should refrain from using fractional points (they are a pain to defend), choose the one one number that best reflects the assignment. For assignments with multiple parts, choose the lowest scoring part.

This rubric is based on the idea that students submit PDF write-ups with their coding assignment. Write-ups *must* be PDF's with the source code so that graders can quickly view them annotate them using blackboard. The rubric does not address specific learning objectives — the assumption is that by completing the assignment the student has implicitly demonstrated some set objectives in addition to coding.

0 points — Student does not submit **all** parts of the assignment, meaning *both* a **PDF** writeup (all sections) that includes source code and output of testing, as well as a .**zip** file with source code.

2 points — The code does not run or does not *appear* to be able to run. The code it much longer than it should be, or does not appear to follow the scaffolding provided. The grader can but **does not have to verify that it does not run**, it is the student's responsibility to provide a writeup that is sufficiently convincing. Student may not appeal by coming after the fact and showing that code runs on their machine.

When grading, the grader should indicate portions of the code by annotating the writeup that are suspicious.

3 points — The code runs or looks like it would run, but the student has not shown via their writeup that it produces the correct result on reasonable inputs. Or, the student has implemented algorithms using approaches other than the ones indicated in the assignment, or the implementation has the wrong asymptotic complexity or that demonstrates a lack of understanding of the assignments objectives. The grader can, but **does not have to run the code to verify correctness** — it is the student's responsibility to make a convincing case that the output and the algorithm is correct.

When grading, the grader should indicate by annotating the write-up where results

4 points — The code runs or appears to run correctly, but has readability or style issues. The student has not demonstrated that their code has passed style guidelines, or the student's implementation appears to be unnecessarily complex (even though it looks like it works).

When grading, the grader should indicate the style problems.

5 points — No issues that we can spot.