One-Dimensional Scrabble

Rules of the game

Scrabble is a well known board game that people of all ages enjoy playing around the world both competitively and socially. Unfortunately, the full two-player game itself with all that branching string matching in two dimensions would be too complicated for the first programming course even in a restricted form, and the full combinatorics of possibilities probably make this game too difficult even for a senior-level course. However, by reducing this game **to a single dimension and turning it from a two player game to a one player puzzle** so that this one player gets to make all the moves enables us to tackle this challenge into submission, armed with only the ideas and techniques that we have learned in this introductory programming course.

One horizontal row of the Scrabble board is given as a Python string guaranteed to consist of characters `'-'` used to denote blank spaces (minus signs are far easier to read and copy from console outputs than whitespace), and English lowercase letters `a-z` so that the row does not yet contain two letters anywhere without at least one blank character between them. One example of such a row of length 30 might be `'--o----v---c---i--b-l--i-n-u-y-'`.

Unlike in the actual Scrabble, you have an unlimited supply of every letter at your disposal. Your task is to replace any of the blanks in the row with letters of your choice to fill that row with words that maximize the overall score for the row. Unless you are lucky enough to find a word that fits the entire row, you still need to leave in some blanks between the words that you construct. Just like in Scrabble, **every maximal block of consecutive letter characters longer than one character must be some legal word** that can be found in our wordlist.

One possible way to fill in the previous row might be `'-convolvulic-trilobal-signeury-'`. If our dictionary is limited to contain only words of maximum length of seven letters, one good legal way to fill in this row might be `'azoxy-everich-bisabol-bignou-y-'`... still coming up with all those "five-dollar words" that your instructor cannot remember ever seeing anywhere.

The returned string is scored by adding up the individual scores for the words inside it. The score of each word is the sum of the Scrabble values of its letters. However, to make this problem more interesting so that you won't just scan for suitable positions to fill with short words of plenty of high-valued letters (such as `'zizz'` or `'xyzzy'`), each word is additionally scored for its length, so that a word with length `k` scores `k*k` points, making longer words significantly more valuable than short words, even if made of common letters worth a point or two each. A word of thirty characters would, by its sheer length, ring in a jackpot of 900 points! However, of course you try to maximize your total score by favouring words made of higher-valued letters if you get to choose between words of equal length.

Tester script

The GitHub repository ikokkari/ScrabbleRow contains the automated tester script scrabblerun.py that you can use to automatically test and evaluate your submission, along with the wordlist file words_sorted.txt. The row patterns given to your function to fill are generated pseudorandomly using the `seed` value defined in the beginning of the tester script. The game is played for the given number of `rounds` for patterns of length `patlen`, with the overall wordlist restricted to use the subset of words in `words_sorted.txt` whose length is between `minlen` and `maxlen`, inclusive.

An example run with settings `seed=7777`, `patlen=50` and `rounds=10` with the instructor's private model solution at the moment of writing this produced the following output. The score for each returned result is displayed in parentheses after the result.

```
scrabblerun with seed=7777, patlen=50 and rounds=10.

Pattern: --h-a-n--u-q--s--u-d-i-e----i-s-i-e--l-e-l-c-t-w-o
Result : mahwa-nabu-qats-quodlibetz-disseize-elve-licht-w-o (360)

Pattern: --s--i--o-u---e--o-t---n--e-s-p-u--u--a-x-e-d-e-g-
Result : assyriologue-teapottykin-zeks-piupiu-falx-exdie-g- (448)

Pattern: -c-h--r-r---e---d--o-n-s---y--h-r-o-i---a--u-p-r-o
Result : scfh-frary-fezzed-cognisably-thyreolingual-upper-o (469)

Pattern: -t---i--e-d--m-i-a---u-u-s-k-n-i---t-a--o---t-o---
Result : athyridae-duumviral-juju-soken-izzat-alloeostropha (496)

Pattern: -r-l-t-l-u-i-n--t-r--s-a-i-e-l----o-c-p-n--e--e---
Result : kral-tellurian-athrepsia-idealize-occupancies-exrx (465)

Pattern: -r-a-s-e-e-p--t-t-b--t-i-e-s-i--d-o-----t-m-e-e--i
Result : krna-skene-phototube-twice-shieldboard-stampeded-i (425)

Pattern: -o-a-i--e-r-e-c---l-v-n-r-a----s-i---y--o-s-n-y-a-
Result : colazione-rheic-folkvangr-aggressively-jobson-yday (479)

Pattern: -i---a-i---o--l-u-o-l-i-i-e-i-m-n--e--u-b-n-a--r-a
Result : piazzalike-oxyl-unoil-isize-immunized-urban-agyria (415)

Pattern: ---u-e-e-n---g-e-i-p-m-p-a-r-k-u--o-i-t-a-u--t-e--
Result : piquiere-nuraghe-i-p-mopla-r-khud-ovist-adulatress (349)

Pattern: -o---a-w--b-e-i-o--i-i-d-n----b-y--o-l-o--h--p--b-
Result : kozuka-womb-erizo-liriodendra-bayamo-loofahs-phoby (410)
4316 Kokkarinen, Ilkka 123456789
```

As you can see in the above solutions, you are allowed to leave in the individual letters already that are inside the pattern without using them if you have no good word available for those positions. Also, even though this possibility does not show up in the solutions above, you are allowed to leave blank sequences longer than one blank in the result, as seen in the instances six, eight and nine. The list line printed by the tester displays the total score followed by the information about the author of the solver. Higher score is better.

For the tester script to be able to find and interact with your code, all your code must be written in a file named `scrabblerow.py`. This file **must contain the following three functions exactly as specified**, plus whatever other helper functions of your own that you choose to write:

```
def author():
```

Returns the name of the author of this script, as string of the form `"Lastname, Firstname"`.

```
def student_id():
```

Returns the student ID of the author of this script as a string.

```
def fill_words(pattern, words, scoring_f, minlen, maxlen):
```

The function receives as parameters the `pattern` as a string and `words` as a list of words whose length is between `minlen` and `maxlen`, inclusive, and a scoring function `scoring_f` that will tell you how many points its argument word will score. From this information, the function returns a new string for the completed row.

The returned result string must be the same length as `pattern`, and in every position where `pattern` has a non-blank letter, the result must contain that same letter. Furthermore, each complete word inside the string must be a legal word from `words_sorted.txt` whose length is between `minlen` and `maxlen`, inclusive. Result strings that do not obey these constraints receive zero points. (The rest of the evaluation will still continue, though.)

Grading the submissions

Each submission is evaluated with `rounds=50` and a secret `seed` value chosen by the instructor. The functions should be fast enough so this test completes in less than five minutes. The submissions are ranked based on their overall scores (the same seed is used for all students). The project grade is computed from this rank in a manner TBA.

It should be easy enough to get a first rudimentary version of this project working, most likely some kind of greedy algorithm that starts filling the pattern from the beginning and tries to find the highest-valued word that can fit into the initial prefix of the pattern. Make sure that this word is followed by a blank in the pattern, and continue after that to fill in the rest of the pattern in the same greedy manner. However, in this kind of thing it often pays not to be too greedy, but put some lower-value word first to leave room for some big jackpot word that could fit later in the pattern…