Generating Crossword Puzzles

AI 2, February 2020

Gabor developed assignment / Eckel developed text

Goal

Write a script to generate American style crossword puzzles. How hard could it be?

The assignment will consist of two (possibly three) parts:

1) Place blocking squares in a crossword puzzle to accommodate an input spec.
2) Determine words that will fill out a crossword puzzle structure.
3) We may have an exciting mystery part if parts 1 & 2 go well!

Advice & Guiding Principles from Last Year

I strongly recommend paying attention to last year's students, who overwhelmingly recommended:

- **DO EXTRA WORK SOONER, NOT LATER.** The *opposite* of procrastination, not merely its absence.
- **PLAN CAREFULLY AND THOROUGHLY.** If you just start coding and see what happens, this is not going to work.
- **WRITE SMALL PIECES / HELPER METHODS AND TEST THEM AS YOU GO.** You can't keep this all in your head at once, I promise. The smaller the pieces are that you can test, the easier it will be to assemble them later.
- **WRITE READABLE CODE.** Name your variables / functions well. Maybe even *comment your code* as you go!
- **DON'T BE AFRAID TO START OVER.** Many students, last year, had to abandon one approach completely and try another. This is a hard choice to make, but if you can't understand your own code, it's a good sign it's time.
- **ASK FOR HELP.** Seriously. Even if you never have before. I *don't* expect you to be able to do this on your own.

If you would like more specific advice, there are 19 pages of it on Blackboard. Everything Mr. Eckel's students wrote last year, only it has been cut & pasted in order to group comments into categories. Every word of what they wrote is in the document.

Part 0 (Getting Started): Input & Output

Command line input to your script will be in the form of:

```
xword.py #x# # dict.txt [seedString1 seedString2 …]
```

- `#x#` is the size of the crossword – first height, then width (eg. 11x15 is 11 high, 15 wide).
- `#` represents the number of blocking squares to be placed.
- `dict.txt` is the name of the dictionary file to use.
- A `seedString` represents placements of fixed items in the crossword. It could be blocks, letters, or a combination. Seed strings have the form of: `H#x#chars` or `V#x#chars`. The leading `H` or `V` indicates horizontal or vertical orientation. The `#x#` indicates the 0-based starting position (where the top left square is `0#0` and the first integer is the vertical offset). Thus, the seed string to place the word "hat" horizontally followed by a blocking square on the top line, starting at the 5th character from the left would be: `H0x4hat#`. It may be that individual letters, word fragments, blocking squares, or a combination are placed rather than complete words. If `chars` is omitted it implies a single blocking square.

For example, this call: `xword.py 11x13 27 wordList.txt H0x0begin V8x12end`

…will generate a crossword puzzle with 11 rows and 13 columns, with 27 blocked squares, using the dictionary in `wordList.txt`, with the word "BEGIN" horizontally from the top left corner and the word "END" going down from the square on row 8 and the last column down into the bottom right square.

For the love of all that is good and holy, **test that this is working before you go any further!**

Output:

Assume that the puzzle in question has *s* squares. The grader is looking for a puzzle or a puzzle structure. A puzzle structure is an unfilled puzzle with just the blocking squares and any seed letters. The grader identifies the puzzle in the output in the following way. From each line, it first removes all whitespace. Then, it removes all lines with any characters in `[^-#a-zA-Z]`. All surviving lines are joined (without newlines) into a single string, and the final *s* characters are extracted.

This way you can print multiple puzzles, and only the final one will be examined, and you can also print debugging statements in the form of `myVar: happyVal` because of the colon, or `Time: 1.234s`.

Part 1: Place Blocking Squares Legally

The first task is to write code that successfully places blocking squares so they follow American crossword puzzle rules.

Specifically:

- Every space on the board must be part of a horizontal word and a vertical word.
- Each word must be at least 3 characters long.
- The blocking squares are symmetric with respect to 180° rotation about the center.
- The non-blocking squares are connected (ie, there cannot be a “wall” of blocking squares, either straight or twisty, separating some spaces/letters from some other spaces/letters).
- Of course, if any command line seed string contains letters, you mayn’t place a blocking square atop any of those letters.

It is worth noting that taking these rules excessively literally can produce amusing results; our grader does take a few test cases to the extreme to make sure you can handle all the possibilities, even the silly ones. For example, this:

```
Your_code.py 6x6 36 wordList.txt
```

…produces a puzzle **completely full of blocks**! This is 100% legal, according to the rules (even if, you know, it’s, like, not very exciting to solve.)

On the AI grader website, the assignment “XWord 1” will accept a script and give it command line arguments as described above. For part 1, you don’t use the dictionary at all, but the argument will still be there for consistency across all parts of the assignment.) You must print out a puzzle that has the requested number of blocking squares, placed legally, and any letters given in the seed strings. You do not need to add any other letters; blocking squares are indicated with **“#”** while the remaining spaces should be blank, indicated by a minus (**“-”**).

Advice for Part 1

Consider these thoughts, as you plan:

- How will you guarantee that the final spaces are all connected?
- How will you deal with a puzzle completely full of blocks, as above?
- How will you deal with a puzzle that begins with two (or more!) disconnected chunks of spaces, requiring some to be filled in?
- The center square on an odd-size board is special. Why? What needs to be in your code as a result?
- For this part of the assignment, you only need to be concerned with whether or not your arrangement of blocking squares is **legal**. However, some boards are clearly better than others for the next part of the assignment. It might be worth reading ahead to consider the advice for future parts when implementing this one.

Part 2: Place Letters to Form Horizontal and Vertical Words

The second task is to successfully place letters in all of the remaining empty spaces so that every horizontal and vertical block of letters forms a valid word.

Specifically:

- A word must be at least three letters long. As you read in the dictionary, remove any words that are too short.
- A word must contain only alphabetic characters. As you read in the dictionary, remove any words that contain non-alphabetic characters (can anyone say "Regular expression"?).
- **EACH WORD MAY ONLY APPEAR IN YOUR CROSSWORD PUZZLE ONCE!**

Advice for Part 2

Consider these thoughts as you plan:

- Is your algorithm going to go **word by word** or **letter by letter**? Both options were able to receive full credit last year, though I have a suspicion word by word is a bit easier. On the other hand, I'm also pretty sure letter by letter is necessary to solve the most difficult one. I'd love to be proven wrong though!
- How are you going to keep track of the locations of each horizontal and vertical word on the board?
- In what order are you going to place words/letters?
  - For instance, you really should not place all the horizontal words first and then check for vertical ones. This will be incredibly inefficient; it will never finish. What would be a better way of deciding which word to fill in next?
  - Similarly, if you're doing letter by letter, I would hesitate to place letters in row major order – left to right, top to bottom. This is likely to result in ridiculous amounts of backtracking. What else is possible?
- How are you going to ensure words are not duplicated? (Ignoring this was a **common mistake** last year!)
- Here are a few simple test cases to get started with, using `twentyk.txt` from blackboard. You should ensure that a complete run of each test case can be done in less than a minute.
  - `Your_code.py 4x4 0 twentyk.txt`
  - `Your_code.py 5x5 0 twentyk.txt V0x0Price H0x4E`
  - `Your_code.py 7x7 11 twentyk.txt`

Specification for Part 2

On the AI grader website, the assignment "XWord 2" will accept a script and give it command line arguments as described above. You'll need to solve each test case in less than a minute. The first eight tests use a dictionary of the 20,000 most common words in English; that dictionary is available on Blackboard as `twentyk.txt` for testing.

Print out a puzzle that has the requested number of blocking squares, placed legally. Any letters given in the seed strings must be present. Every horizontal and vertical block of letters must be a **distinct** word in the given dictionary.

Considerations

For part 2, your part 1 code should first place blocking squares, *and* then place the words after that.

This means that your code will need to place the blocking squares *intelligently*, not simply in *any* manner that follows the rules. There are more small words than large ones, and the more letters you're trying to match the less likely it is that a word will exist that fits it perfectly. As a result, it is prudent to maximize the number of small words on your grid.

In other words, this is good…

```
- - - - - # # - - - - - -
- - - - - # # - - - - - -
- - - - - - # - - - - - -
- - - - # - - - - # - - -
- - - # - - - - # - - - -
- - - # - - - # - - - - -
# # # - - - # - - - # # #
- - - - - # - - - # - - -
- - - - # - - - - # - - -
- - - # - - - - # - - - -
- - - - - - # - - - - - -
- - - - - - # # - - - - -
- - - - - - # # - - - - -
```

…but this is bad.

```
# # - - - - - - - - - # #
# # - - - - - - - - - # #
# # - - - - - - - - - # #
- - - - - - - - - - - - -
- - - - - - - - - - - - -
- - - - - - - - - - - - -
# # - - - - # - - - - # #
- - - - - - - - - - - - -
- - - - - - - - - - - - -
- - - - - - - - - - - - -
# # - - - - - - - - - # #
# # - - - - - - - - - # #
# # - - - - - - - - - # #
```

If it's not clear to you why this is important, just try running the test cases below without optimizing for blocking square position and you'll figure it out pretty quickly…

Further advice

Consider these thoughts as you plan:

- How can you identify what would be a good space to place a blocking square? How can your code prioritize those spaces?
- With a larger number of spaces to work with, do you need to also change your algorithm for placing letters or words?
- Here are three excellent test cases to use to test this part of the code. Remember you'll need to solve each of these in less than a minute. These use `wordlist.txt`, a dictionary of actual crossword puzzle answers.
    - `Your_code.py 9x13 19 wordlist.txt V0x1Dog`
    - `Your_code.py 9x15 24 wordlist.txt V0x7con V6x7rum`
    - `Your_code.py 13x13 32 wordlist.txt V2x4# V1x9# V3x2# h8x2#moo# v5x5#two# h6x4#ten# v3x7#own# h4x6#orb# h0x5Easy`

- The next test case is second tier in difficulty on this assignment. Your code should be able to start completely from scratch with a blank board of size 13x13 with a relatively small number of blocking squares and generate a puzzle in under a minute. The example boards above are possible outputs for this test case; see how close your code can get to a configuration with the same characteristics as the example on the left.
    - `Your_code.py 13x13 29 wordlist.txt`

- Finally, to get back to the real-world connection here, I present two more interesting test cases. These are boards with the structure completely specified to match actual published crosswords. The first is empty, the second has a few seed words. If your code can solve these in less than a minute, you're doing well:
  - `Your_code.py 15x15 42 wordlist.txt H0x0# V0x7### H3x3# H3x8# H3x13## H4x4# H4x10## H5x5# H5x9## H6x0### H6x6# H6x10# H7x0##`
  - `Your_code.py 15x15 42 wordlist.txt H0x0#MUFFIN#BRIOCHE V0x7## H3x3# H3x8# H3x13## H4x4# H4x10## H5x5# H5x9## H6x0### H6x6# H6x10# H7x0## H14x0BISCUIT#DANISH`

Finally …

The goal here is to see how far you can optimize your code.

As it happens, in general, an NxN board with N^2 / 4 blocking squares is reasonably easy to generate. N^2 / 6 is harder, but closer to actual for-real published puzzle standards. Let's try to play with these standards at larger sizes.

You can expect a final test case or cases where six random letters will be taken from the set {E, T, A, O, I, N, S, H, R, D, L, U} and they will be put at random locations on the board. There may be up to three sizes:

- A 20x20 board with 66 blocking squares (N^2 / 6).
- A 25x25 board with 156 blocking squares (N^2 / 4).
- A 25x25 board with 104 blocking squares (N^2 / 6).