Generating Crossword Puzzles

Eckel, TJHSST AI1, Fall 2022

Assignment

Just write a program to generate crossword puzzles.

How hard could it be?

Advice & Guiding Principles from Previous Students

I strongly recommend paying attention to prior 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's 14 pages of it on the course website. I coped everything my students wrote from the first two years we assigned this, only cutting / pasting to group comments into categories. Every word.

Part 0 (or, Getting Started): Input & Output

You'll read in the name of a text file on the command line. That file will have one challenge per line in this format:

```
Yourfile.py dict.txt #x# # (seedstrings)
```

Here is what all the arguments represent:

- `dict.txt` is the name of the dictionary (word list) file used. (You should just ignore this argument for part 1.)
- `#x#` represents the size of the crossword puzzle, **first height** then width. (Or, think of it as rows then columns.)
- `#` represents the number of blocked squares you must place.
- `(seedstrings)` stands for any number (including zero) of strings of characters. Each string is formatted as follows – `H*x*characters` or `V*x*characters`. H or V stand for horizontal or vertical, and *x* refers to two numbers separated by an x. These numbers locate the first character of the string. The first number is the string's row, zero-indexed, and the second number is the string's column, zero-indexed. You must place the given string at the given location progressing in the given direction one character at a time. Please note: *this will not necessarily be a complete word, and you should **not** automatically place blocking squares on either end of it. It may, in fact, contain blocking squares*.

For example, this call: `Your_code.py wordlist.txt 11x13 27 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!**

In terms of output, while working on this assignment you should print the board nicely so you can read the output. At the very end, I'll ask for output in a single string, but don't worry about that now; make output you can read easily.

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.
- Every word must be at least 3 characters long.
- The board must be 180 degree rotationally symmetric.
- All of the spaces must form one connected block (ie, there cannot be a "wall" of blocking squares, either straight or twisty, separating some spaces from some other spaces).
- Of course, if any of the command line seed strings contain letters, you can't place a blocking square on top of any of those letters.

It is worth noting that taking these rules 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:

```
nodict.txt 6x6 36
```

…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.)

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 is **legal**. However, some boards are clearly better than others for later parts of the assignment. It might be worth skipping ahead to part 3 to read the advice there and consider it in advance.

Test Your Code

The test file CrosswordTests1.txt contains 11 test cases for part 1.

You should make sure your code can complete the full test file in less than 2 minutes (you'll need much faster than that for the later parts, but if you're just going for GREEN I'll take 2 minutes).

While you're testing, print out the output nicely each time and use your eyes and your brain to make sure your output is right. (In other words, check to make sure you end up with a symmetric, connected puzzle with the right number of blocks.)

Any letters specified in the puzzle seed strings should be present, but you shouldn't solve any other letters; use a _ for any blank space on the board.

Some Part 1 Example Output

Here are two of the test cases in the .txt file. For each one, there is the input string from the .txt file, then a board with ONLY the information from the input string, and finally a correct final answer. **These are not the only possible correct final answers**, just one example.

```
nodict.txt 13x13 32 H1x4#TOE# H9x2# V3x6# H10x0SCINTILLATING V0x5STIRRUP H4x2##ORDAINED V0x1NOPE V0x12LAW V5x0ENT

- N - - - S - - - - - - L
- O - - # T O E # - - - A
- P - - - I - - - - - - W
- E - - - R # - - - - - -
- - # # O R D A I N E D -
E - - - - U - - - - - - -
N - - - - P - - - - - - -
T - - - - - - - - - - - -
- - - - - - - - - - - - -
- - # - - - - - - - - - -
S C I N T I L L A T I N G
- - - - - - - - - - - - -
- - - - - - - - - - - - -

- N - - # S - - # - - - L
- O - - # T O E # - - - A
- P - - - I - - - - - - W
- E - - - R # - - - # # #
# # # # O R D A I N E D #
E - - # - U - # - - - - -
N - - - # P - - # - - - -
T - - - - # - - - # - - -
# - - - - - - - - # # # #
# # # - - - # - - - - - -
S C I N T I L L A T I N G
- - - - # - - - # - - - -
- - - - # - - - # - - - -

nodict.txt 9x20 32 H4x7D# H3x4R# H2x4# V2x0EAT V5x1#I V8x16C

- - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - -
E - - - # - - - - - - - - - - - - - -
A - - - R # - - - - - - - - - - - - -
T - - - - - - D # - - - - - - - - - -
- # - - - - - - - - - - - - - - - - -
- I - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - C - -

- - - - # - - - # - - - # - - - # - -
- - - - # - - - - - - - # - - - - - -
E - - - # - - - - - - - # - - - - - -
A - - - R # - - - - - - - # - - - # # #
T - - # - - - D # # # # - - - - # - - -
# # # - - - # - - - - - - - # - - - - -
- I - - - - - # - - - - - - - # - - - -
- - - - - - - # - - - - - - - # - - - -
- - - # - - - # - - - # - - - # C - - -
```

Specification for GREEN Credit for Crosswords: Build the Grid

Submit a single python script to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code accepts a single command line argument – the name of a txt file of puzzles as specified on page 1.
- **Your output should not look like the sample runs above.** For each puzzle, you output **a single line of output and nothing else.** (If the input file has 6 puzzles, you should print 6 lines, no more and no less.) Each line should be a full crossword board as specified in row major order with no extra spaces.
- Runtime is less than 2 minutes. (Comparable to the full sample puzzle file I provided.)

Part 2: Place Letters to Form Horizontal and Vertical Words

The second task is to write code that successfully places 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 (the Python function `isalpha()` might be useful here).
- **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 get black credit. 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 is a **common mistake**!)

Test Your Code

The test file CrosswordTests2.txt contains 11 test cases for part 2.

You should make sure your code can complete the full test file in less than 2 minutes.

While you're testing, print out the output nicely each time and use your eyes and your brain to make sure your output is right. (In other words, check to make sure you end up with a symmetric, connected puzzle with the right number of blocks, then look at the words to make sure that they don't repeat. On puzzles this small, you can definitely verify this by eye/by hand.) Printing out the starting information like I did in my examples on the previous page is a good way to make sure the seed strings are all in the right place too.

Specification for BLUE Credit for Crosswords: Fill the Grid

Submit a single python script to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code accepts a single command line argument – the name of a txt file of puzzles as specified on page 1.
- For each puzzle, you output **a single line of output and nothing else.** (If the input file has 6 puzzles, you should print 6 lines, no more and no less.) Each line should be a full crossword board as specified in row major order with no extra spaces, just like part 1. But unlike part 1, the crossword should contain no blanks!
- Runtime is less than 2 minutes. (Comparable to the full sample puzzle file I provided.)

Part 3: Generate Complete Puzzles

Finally, combine parts 1 and 2 and get them working together so you can place the blocking squares *and* 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, you want 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 third test file without optimizing for blocking square position and you'll figure it out pretty quickly…

Advice for Part 3

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?

Test Your Code

The test file CrosswordTests3.txt contains 10 test cases for part 3.

Just for fun, the final test case is actually a published crossword puzzle with four words already filled in – this is exactly the kind of thing you'd need to make if you were actually professionally making crossword puzzles!

This test file is new this year. I think you should be able to run each test case in less than 30 seconds. If it turns out you can run all of them in less than 30 seconds except one or two, let me know – it's possible I need to adjust.

Specification for RED credit for Crosswords: Create the Entire Puzzle

Submit a single python script to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code accepts a single command line argument – the name of a txt file of puzzles as specified on page 1.
- For each puzzle, you output **a single line of output and nothing else.** (If the input file has 6 puzzles, you should print 6 lines, no more and no less.) Each line should be a full crossword board as specified in row major order with no extra spaces, just like part 1. But unlike part 1, the crossword should contain no blanks!
- Runtime is less than 30 seconds per puzzle. (Comparable to the full sample puzzle file I provided.)

Specification BLACK credit for Crosswords: Optimization

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 standard. Let's try to play with these standards at larger sizes.

I will generate three test cases randomly by taking six letters from the set {E, T, A, O, I, N, S, H, R, D, L, U} and putting them at random locations on the board. There will be 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 ($N^2 / 6$).

You have three more test files, each one has 10 examples of one of the above options.

If you can solve all 10 puzzles in any one file in less than one minute each, you're ready.

Submit a single Python script that takes the usual command line arguments to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your code successfully completes **any one** of the three tests above in less than a minute.