Generating Crossword Puzzles

Eckel, TJHSST AI2, Spring 2021

Assignment

Just write a program to generate crossword puzzles.

How hard could it be?

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's 14 pages of it on Blackboard. I coped everything my students wrote two years ago, only cutting / pasting to group comments into categories. Every word of what they wrote is in the document.

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

Input will be via command line inputs of the form:

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

Here is what all the arguments represent:

- `#x#` represents the size of the crossword puzzle, **first height** then width.
- `#` represents the number of blocked squares you must place.
- `dict.txt` is the name of the dictionary (word list) file used.
- `(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# locates 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 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 is straightforward. The grader will accept either a single long string or an output that is printed in actual rows with newlines at the end and any amount of space characters anywhere. Use "#" for blocking squares and "-" for blanks.

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:

```
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.)

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.

Specification for Part 1

On the AI grader website, the assignment "XWord 1" will accept a script and give it command line arguments as described above. (Note that for part 1, you don't use the dictionary at all, but the argument will still be there so that input/output is consistent across all of the assignments.)

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; the remaining spaces should be blank ("-").

Once you have passed all 10 test cases on the grader, submit a screenshot to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- You submit a screenshot demonstrating you have passed all 10 test cases.

For **resubmission**:

- Complete the specification correctly.

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 Outstanding Work. 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 the course website. 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 the course website 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.

Credit for part 2 is given **when you pass the first 9 test cases on the grader.** (Not all 11.) Once you have done so, submit a screenshot to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- You submit a screenshot demonstrating you have passed **the first 9** test cases. (Not all 11.)

For **resubmission**:

- Complete the specification correctly.

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 test cases below 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?
- 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 approximately the most difficult thing your code should be able to do for full credit 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 golden:
    - `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`

Specification for Part 3

On the AI grader website, return to the assignment "XWord 2" and complete the **last two test cases**. As before, you'll have one minute per test case. These test cases use the Crossword dictionary available on Blackboard as `wordlist.txt`.

After you've passed both test cases, submit **a screenshot AND your code** to the final link on the course website. I'll have a couple more test cases to run, both of which I expect to take longer than a minute (which makes them a poor fit for the online grader – I don't want to take up too many resources).

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- You submit a screenshot demonstrating you have passed **the last two test cases** on Xword 2.
- You also **submit your code** that has accomplished this feat.

For **resubmission**:

- Complete the specification correctly.

Specification for 1 or 2 Outstanding Work Credits: Optimize!

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 ($N^2 / 6$).

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

This assignment receives outstanding work credit if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- If your code successfully completes **any one** of the three tests above in less than a minute, you will receive **ONE** outstanding work credit.
- If your code successfully completes **all three** of the three tests above in less than a minute each, you will receive **TWO** outstanding work credits.

For **resubmission**:

- Complete the specification correctly.