ECS 32B: Programming Assignment #2

Instructor: Aaron Kaloti

Summer Session #2 2020

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Made it clearer in part #3 the number of whitespace in certain portions (see red text).

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Saturday, August 29. Gradescope will say 12:30 AM on Sunday, August 30, due to the "grace period" (as described in the syllabus). *Rely on the grace period for extra time at your own risk.*

Some students tend to email me very close to the deadline. This is a bad idea. There is no guarantee that I will check my email right before the deadline.

*This content is protected and may not be shared, uploaded, or distributed.

3 Grading Breakdown

The assignment is out of 160 points. Here is a *tentative* breakdown of the worth of each part.

- Part #1: 15
- Part #2: 20
- Part #3: 25
- Part #4: 15
- Part #5: 30
  - Part #5.1: 5
  - Part #5.2: 25
- Part #6: 20
- Part #7: 35
  - Part #7.1: 20
  - Part #7.2: 15

4 Gradescope Autograder Details

Once the autograder is released, I will update this document to include important details here. Note that you should be able to verify the correctness of your functions without depending on the autograder. I am aiming to work on the autograder around 08/22.

4.1 Released Test Cases vs. Hidden Test Cases

You will not see the results of certain test cases until after the deadline; these are the hidden test cases, and Gradescope will not tell you whether you have gotten them correct or not until after the deadline. One purpose of this is to promote the idea that you should test if your own code works and not depend solely on the autograder to do it. Unfortunately, Gradescope will display a dash as your score until after the deadline, but you can still tell if you have passed all of the visible test cases if you see no test cases that are marked red for your submission.

4.2 Released Test Cases' Inputs

TBA.

4.3 Changing Gradescope Active Submission

Once the deadline occurs, whatever submission is your **active submission** will be the one that dictates your final score. By default, your active submission will be your latest submission. However, you can change which submission is your active submission, which I talk about a small video that you can find on Canvas here. This video is from my ECS 32A course last summer session, so the autograder referenced therein is for a different assignment. Note that due to the hidden test cases, your score will show as a dash for all of the submissions, but you can still identify how many of the released test cases you got correct for each submission.

5 Ayayron™: An Editor That No One in Their Right Mind Would Ever Use

In this assignment, you will implement a text editor. The only actual editing that this editor will permit is insertion of a phrase on a given line (based on where the cursor is) and the trimming of trailing whitespace. Other helpful operations will include movement of the cursor, scrolling, and saving to a file.

Your entire submission will be in a file called `editor.py`. You can find a skeleton version of `editor.py` and a complete implementation of the `Stack` implementation from lecture in `stack.py` on Canvas. You can change the parameter names of the methods in `editor.py`, if you wish. You will only submit `editor.py` to the autograder; the autograder will use its own copy of `stack.py`, meaning that you are not allowed to modify this file. So far, the code should just be able to tell you when you've entered an invalid command and should allow you to quit.

```
>>> e = Editor('ignore', 'ignore')
>>> e.run()
Enter command: d

Enter command: a
```

```
Enter command: z
Invalid command.

Enter command: o
Invalid command.

Enter command: end
>>>
```

Although the parts below come in a certain order, you don't have to do them in the exact order shown. Needless to say, some parts have to be done (or at the very least, *should* be done) before others; for example, it wouldn't make much sense to implement the window scrolling before you implement the reading of the input file's contents. I personally found part #5.2 to be the hardest part. In some autograder cases, I may try to reduce the dependency of certain parts on each other. For example, even if you cannot quite get part #5.2 fully working, it will be possible to get some points for parts #6 and #7.

Although you understandably may not understand every single detail of these specifications on your first read through, you may find it worthwhile to read the entire specifications before starting anyways.

I would recommmend skimming the list of string and list methods here and here. I am not prohibiting any of these on this assignment. I personally found the string methods `rstrip()` and `rjust()` to be useful.

Do not forget to close any file that you open, once you are done using it. Or just use the `with` operator.

You will find that you'll need to add members to the `Editor` class that are not explicitly stated in the direcitons; you can figure these out as you go.

5.1 Part #1 - Loading Keys: `load_keys()`

As you can see in `editor.py`, there are default values of each of the key members, from `self.down_key` to `self.redo_key`. Each such member contains the key that must be entered in order to do that operation; for example, to undo the last edit/insertion, the user would need to press the key whose value is in `self.undo_key`.

The last parameter (or, you could say, third explicit parameter), `settings_filename`, is the name of a settings file. This file is used in order to override the default keys. It is not always the case that one wants to override the default keys, which is why the `settings_filename` has a default value of `None`.

Your task for this part is to implement the `load_keys()` method. If properly formatted, the settings file should have eight lines, with one character per line. In order, the characters should be the following:

1. Down key.
2. Up key.
3. Left key.
4. Right key.
5. Insert key.
6. Save key.
7. Undo key.
8. Redo key.

You may assume that the file always has eight lines. The `load_keys()` method should `raise` a `ValueError` if any of the lines is empty or contains more than one character (ignoring the newline character).

This next detail is something that I hate to waste a paragraph on, but it is worth pointing out because it relates to potential bugs you may encounter. You may recall that each line read from the file will end with a newline character. Unfortunately, some editors (e.g. Sublime Text) do not automatically insert a newline character at the end of the last line of a text file. This is arguably a bug/flaw in such editors[1], and you should expect that any text file used by the autograder will have a newline character at the end of its last line. However, if you create your own text files and find that your code crashes on the last line, then it could be because of the lack of a newline character at the end of the last line, depending on which editor you used.

I personally found it easier to first check if any line in the file has the wrong length and then read the keys. This required two passes through the file, causing me to use the `seek()` method (with `0` as the argument) to tell the file object to start from the beginning of the file. Obviously, doing two passes is arguably inefficient, and you don't need to do that if you don't want to; you don't need to use `seek()`.

Below is are examples (along with relevant text files) of what the result of `load_keys()` should be.

`settings1.txt`

[1] I say this because the POSIX standards define a text file as a file organized into lines, where each line ends in a newline character. This is as opposed to a binary file (which admittedly may seem like nonsense, because this definition implies that a text file that does not end in a newline character should be called a binary file even if it clearly only contains human-readable text, but that's beside the point).

```
d
u
l
r
i
s
z
y
```

bad_settings1.txt

```
d
u
l
rr
i
s
z
y
```

bad_settings2.txt

```
d
u
l

i
s
z
y
```

```
>>> e = Editor("ignore for now", "ignore this for now too")  # default (no third argument)
>>> e.down_key
's'
>>> e.redo_key
'r'
>>> e = Editor("ignore for now", "ignore this for now too", "settings1.txt")
>>> e.down_key
'd'
>>> e.redo_key
'y'
>>> e.up_key
'u'
>>> e = Editor("ignore for now", "ignore this for now too", "bad_settings1.txt")
Traceback (most recent call last):
...
ValueError: At least one line in settings is too long.
>>> e = Editor("ignore for now", "ignore this for now too", "bad_settings2.txt")
Traceback (most recent call last):
...
ValueError: At least one line in settings is empty.
```

5.2 Part #2 - Reading the Edited File's Contents

The first explicit argument to the `Editor` initializer, called `infilename`, is the name of a file whose contents the editor will start with when editing. You need to read and store this file's contents. There are a few options that you have for storing the file's contents. The most intuitive (and probably best) option is a list of strings. Another option is a list of lists of characters. The primary benefit of a list of lists of characters is that you will not have to deal with the fact that strings are immutable in part #6, but other than that, a list of strings is perhaps preferable. If you go with the list of lists of characters, you may find the fact that `list(...)` can convert a string into a list of its characters to be useful, as shown below:

```
>>> list("abcd")
['a', 'b', 'c', 'd']
```

Your initializer (or a helper function called by the initializer) should `raise` a `ValueError` in either of the following cases:

- At least one line in the file is longer than the window width (see part #3), which is 20 characters (not counting the newline character).
- The input file is empty.
- The input file has more than 30 lines.

No example output is shown, since part #3 indirectly does this. As stated in parts #3 and #4, the editor *trims* trailing whitespace (see part #3 for a description of what this means), so you may want to make use of the string method `rstrip()` in this part.

5.3 Part #3 - Printing the Current File Contents: `print_current()`

Whenever the edited file's contents are displayed, it will be within a window that is 20 characters wide and 10 characters tall. (These values 20 and 10 are stored in the `window_width` and `window_height` fields of the `Editor` class, as you can see at the start of the initializer.) Through this window, we see only a part of the edited file's contents at any given time. The file's contents will never be wider than the window, because the `Editor` initializer would have raised a `ValueError` and, as you'll see in part #6, the user will not be allowed to extend a line so that it's wider than the window[2]. Initially, the window should start at line 1; in part #5.2, you will allow the window to be scrolled down. For example, suppose that we have the following text file.

```
abcdef
ab

xyz
Hi there
Hi how are you
I am good
blah
blah blah blah blah
stream of
conciousness
don't fail me now
blah

that is all
```

Without the window being moved, `print_current()` would result in only the first 10 lines being printed, as shown below.

```
     *
     12345678901234567890
*  1 abcdef
   2 ab
   3
   4 xyz
   5 Hi there
   6 Hi how are you
   7 I am good
   8 blah
   9 blah blah blah blah
  10 stream of
     12345678901234567890
```

(Update: Just so it's clear, there are two whitespaces between the asterisk and the number 1, in the case of the asterisk on the left column. If the asterisk were on line number 10, then there would be one whitespace between the asterisk and the number 10.)

Below is the above, but shown on the Python IDLE interpreter instead. This is what it should look like so far if you have completed this part. Note that the `run()` method already calls `print_current()` when appropriate. In fact, I would *highly recommend against* modifying `run()` unless you have a good reason to do so. You do not need to store the text files that you use in a folder called `text_files`; I did that for my own file organization.

```
>>> e = Editor('text_files/input3.txt', 'ignore for now')
>>> e.run()
     *
     12345678901234567890
*  1 blah
   2 blah blah
   3 blah blah blah
   4
   5 what is up
   5
   6
   7
   8
   9
  10
```

[2] As I said above, no one would want to use this editor.

```
    12345678901234567890
Enter command:
```

As you can see, rows containing `12345678901234567890` are printed above and below the window's contents, for convenience. (It is sometimes useful to know which column you are on in an editor, even if that editor is terrible.) You can also see two asterisks in the above output as well. Those asterisks mark the current location of the cursor. In part #5, you will allow the cursor to be moved, and in part #6, you will allow the user to make insertions starting at the current location of the editor. Before getting to those parts, you can always manually change the position of the cursor in the code in order to check if your `print_current()` implementation works regardless of the cursor's location, so as to minimize the chance that you have to modify `print_current()` once you are done with this part. Lastly, the line numbers of the file are shown. The max line number is 30 (since files that are longer than that are rejected by the `Editor` initializer). Since line numbers may be one digit or two digits, you need to be mindful of how many whitespaces to print before it; this is why I suggest the `rjust()` string method.

Regarding trailing whitespaces: "Trailing whitespace" refers to the occurrence of whitespace at the end of a line. When printing strings out, it is difficult to visually identify trailing whitespace. For example, for all you know, there could be 30 blank whitespaces after the asterisk printed in the first line of the above example output. **Do not worry about trailing whitespace in this assignment.** The autograder will ignore trailing whitespace when comparing the output of your code with mine. In some cases, part of the file itself may include trailing whitespace. For example, line #1 in the above example output could have four whitespaces after the `abcdef`, and line #3 could contain six whitespaces in it. You do not need to worry about trailing whitespace here either. **The editor will *trim* trailing whitespace**, meaning that all trailing whitespace will be removed, e.g. line #1 would have the four trailing whitespaces removed, and line #3 would be reduced to a completely empty line. This primarily matters for part #4.

Suppose that in the above example, the window instead was currently starting at line #4 and the cursor was at the sixth column and the eighth row. In that case, the output of `print_current()` would look like below. Again, it will not be possible to move the cursor or scroll the window until you've done part #5. However, you may want to experiment with moving the window artificially (by changing the relevant members that you hopefully will add to your `Editor` class) before moving on from this part.

```
          *
    12345678901234567890
   4 xyz
   5 Hi there
   6 Hi how are you
   7 I am good
   8 blah
   9 blah blah blah blah
  10 stream of
* 11 conciousness
  12 don't fail me now
  13 blah
    12345678901234567890
```

Below is an example (the file and the initial view) of what happens when the edited file lacks enough lines to fill the entire height of the window.

File:

```
blah blah
blah blah blah

what is up
```

Initial output of `print_current()`:

```
    *
    12345678901234567890
*  1 blah
   2 blah blah
   3 blah blah blah
   4
   5 what is up
   6
   7
   8
   9
  10
    12345678901234567890
```

5.4 Part #4 - Saving the Current File's Contents: `save()`

Implement the `save()` method so that it saves the contents of the currently edited file. The file that you should save to is the one whose name was given as the second explicit argument in the initializer of `Editor`. Its original contents should be erased. Note that the same file can be the one that was read and written to (in which case the first and second arguments to the `Editor` initializer would both be that file's name).

This method is already called by the `run()` method when the user enters the save key.

If you chose to store the edited file's contents in a list of lists of characters, then you may find the string method `join()` to be useful, as shown below.

```
>>> "".join(['a', 'b', 'c', 'd'])
'abcd'
```

Below is an example of how saving should behave.

Input file (first argument to initializer):

```
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
```

Interpreter interaction:

```
>>> e = Editor('text_files/input4.txt', 'text_files/output4.txt')
>>> e.run()
     *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: e

     *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: q
>>>
```

Resulting output file (second argument to initializer), which is exactly the same as the input file since no modifications were made:

```
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
```

```
Line 8
Line 9
Line 10
Line 11
Line 12
```

As stated before, the editor removes trailing whitespace. Thus, if the initial file contains trailing whitespace, the output file will *not* contain it. I would deal with this by using the string method `rstrip()` while writing out each line to the output file.

Input file: (there are three whitespaces after "Line 3")

```
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
```

Interpreter interaction:

```
[Basically the same as in the previous example.]
```

Resulting output file:

```
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
```

There is a file method called `writelines()` that may eliminate the need for a loop in this part. In my own implementation, I didn't use `writelines()`, since I chose not to store the newline character at the end of each line.

5.5 Part #5 - Moving the Cursor

5.5.1 Part #5.1 - Leftward/Rightward Movement: `move_left()` and `move_right()`

The `move_left()` and `move_right()` methods should change the "x-coordinate" of the cursor, i.e. change which column the cursor is on. If moving the cursor would cause it to go out of bounds (i.e. to before the first column or after the 20th column), then the movement should be prevented.

Below are examples of how your code should behave once you have successfully implemented `move_left()` and `move_right()`. Note that the `run()` method already calls these methods when the appropriate directional commands are entered.

```
>>> e = Editor('text_files/input1.txt', 'ignore')
>>> e.run()
     *
    12345678901234567890
*   1 abcdef
    2 ghi
    3 jklm
    4 jkj
    5 jklasj
    6 jklajs;lkfj
    7 qiowuioj
    8 lkjw;lkj
    9 qwklejklj;c
   10 jkaljsoiu
    12345678901234567890
Enter command: d

```

```
      *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
   6 jklajs;lkfj
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: d

       *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
   6 jklajs;lkfj
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: d

        *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
   6 jklajs;lkfj
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: d



[... omitting some parts in order to save electronic trees ...]



                       *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
   6 jklajs;lkfj
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: d

                        *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
   6 jklajs;lkfj
```

```
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: d

                        *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
   6 jklajs;lkfj
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: d

                        *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
   6 jklajs;lkfj
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: d

                        *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
   6 jklajs;lkfj
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: d

                        *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
   6 jklajs;lkfj
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: a

                       *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
```

```
   6 jklajs;lkfj
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: q
>>>
```

5.5.2 Part #5.2 - Upward/Downward Movement and Scrolling the Window: `move_up()` and `move_down()`

As I said previously, I personally found this part to be the hardest (although that might be because I had to come up with all of the below rules by myself), and there will be test cases in parts #6 and #7 that won't involve upward/downward cursor movement, meaning that you can still get those test cases correct even if you do not finish this part.

When no scrolling occurs, up/down movement is simple: move the cursor up or down. When scrolling occurs, however, then the rules are more complicated. The following rules dictate how scrolling up and down should work. I based these rules off of Sublime Text.

- **Scrolling up**: **Scrolling up can only occur when the user tries to move the cursor up while the cursor is at the top of the window.** Moreover, if the cursor is already at the first line of the file, then scrolling cannot occur; otherwise, scrolling should proceed, with the cursor staying at the top of the window.
- **Scrolling down. Scrolling down can only occur when the user tries to move the cursor down while the cursor is at the *last line of the file*.** This difference means that you can view past the end of the file when scrolling down (although you cannot edit past the end of the file), but you can't view before the start of the file when scrolling up. Here are four scenarios.
  1. If the cursor is *not* at the last line of the file but *is* at the bottom of the window, then scroll down (without moving the cursor).
  2. If the cursor is at the last line of the file but *not* at the top of the window, then scroll down but move the cursor *up*. This will simulate the effect of scrolling down while keeping the cursor at the same position relative to the file's contents.
  3. If the cursor is at the last line of the file *and* at the top of the window, then scrolling down should be prohibited. In other words, although the user can scroll past the end of the file, they cannot cause the last line of the file to completely disappear. (Recall from part #2 that the `Editor` initializer rejects empty files, so you need not worry about that case.)

Below is an example of how your code should behave after you have completed this part.

Input file:

```
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
```

Interpreter interaction:

```
>>> e = Editor('text_files/lines.txt', 'ignore')
>>> e.run()
     *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
```

```
     12345678901234567890
Enter command: w

     *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: d

      *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: w

      *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   1 Line 1
*  2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: w

      *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
```

```
  10 Line 10
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   1 Line 1
*  2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   1 Line 1
   2 Line 2
*  3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: s



[... omitting some parts in order to save electronic trees ...]



      *
     12345678901234567890
   1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
*  8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
*  9 Line 9
  10 Line 10
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   1 Line 1
   2 Line 2
```

```
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
* 10 Line 10
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
* 11 Line 11
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
* 12 Line 12
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
* 12 Line 12
  13
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
* 12 Line 12
  13
  14
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   6 Line 6
```

```
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
* 12 Line 12
  13
  14
  15
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
* 12 Line 12
  13
  14
  15
  16
     12345678901234567890
Enter command: w

      *
     12345678901234567890
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
* 11 Line 11
  12 Line 12
  13
  14
  15
  16
     12345678901234567890
Enter command: w

      *
     12345678901234567890
   7 Line 7
   8 Line 8
   9 Line 9
* 10 Line 10
  11 Line 11
  12 Line 12
  13
  14
  15
  16
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
* 11 Line 11
  12 Line 12
  13
  14
  15
  16
     12345678901234567890
Enter command: s

      *
     12345678901234567890
```

```
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
* 12 Line 12
  13
  14
  15
  16
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
* 12 Line 12
  13
  14
  15
  16
  17
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   9 Line 9
  10 Line 10
  11 Line 11
* 12 Line 12
  13
  14
  15
  16
  17
  18
     12345678901234567890
Enter command: s

      *
     12345678901234567890
  10 Line 10
  11 Line 11
* 12 Line 12
  13
  14
  15
  16
  17
  18
  19
     12345678901234567890
Enter command: s

      *
     12345678901234567890
  11 Line 11
* 12 Line 12
  13
  14
  15
  16
  17
  18
  19
  20
     12345678901234567890
Enter command: s

      *
```

```
     12345678901234567890
* 12 Line 12
  13
  14
  15
  16
  17
  18
  19
  20
  21
     12345678901234567890
Enter command: w

      *
     12345678901234567890
* 11 Line 11
  12 Line 12
  13
  14
  15
  16
  17
  18
  19
  20
     12345678901234567890
Enter command: w

      *
     12345678901234567890
* 10 Line 10
  11 Line 11
  12 Line 12
  13
  14
  15
  16
  17
  18
  19
     12345678901234567890
Enter command: w

      *
     12345678901234567890
*  9 Line 9
  10 Line 10
  11 Line 11
  12 Line 12
  13
  14
  15
  16
  17
  18
     12345678901234567890
Enter command: w

      *
     12345678901234567890
*  8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
  12 Line 12
  13
  14
  15
  16
  17
     12345678901234567890
Enter command: s

```

```
      *
     12345678901234567890
   8 Line 8
*  9 Line 9
  10 Line 10
  11 Line 11
  12 Line 12
  13
  14
  15
  16
  17
     12345678901234567890
Enter command: s

      *
     12345678901234567890
   8 Line 8
   9 Line 9
* 10 Line 10
  11 Line 11
  12 Line 12
  13
  14
  15
  16
  17
     12345678901234567890
Enter command: w

      *
     12345678901234567890
   8 Line 8
*  9 Line 9
  10 Line 10
  11 Line 11
  12 Line 12
  13
  14
  15
  16
  17
     12345678901234567890
Enter command: w

      *
     12345678901234567890
*  8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
  12 Line 12
  13
  14
  15
  16
  17
     12345678901234567890
Enter command: w

      *
     12345678901234567890
*  7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
  12 Line 12
  13
  14
  15
  16
     12345678901234567890
Enter command: w
```

```
      *
     12345678901234567890
*  6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
  12 Line 12
  13
  14
  15
     12345678901234567890
Enter command: w

[... etc. ...]
```

5.6 Part #6 - Insertion: `insert()`

The `insert()` method should insert its argument at the appropriate part of the file contents, based on where the cursor is. See the below examples. (Note that this should not modify the output file at all yet; to be clear, if the user never enters the save command, then the output file is ignored.)

The `run()` method already detects if the user enters the insert command, and if the user does, the `run()` method parses whatever is after the insert command and sends it to the `insert()` method.

If the insertion is successful, then the `insert()` method should return `True`. If the insertion would extend past the end of the current line (the one that the cursor is on), then the function should return `False`; *you should avoid changing the file's currently depicted contents in this case*. Note that writes that span multiple lines are not supported in this editor, which also means that you cannot add lines. You *can*, however, extend an already-existing line (not past the width of the window, of course).

Hint: You may want to read part #7 before starting this part, because parts #6 and #7 are linked such that you may find you have to go back and modify your `insert()` method in order to do part #7.

Below is an example of how your code should behave after finishing this part. This example does not involve cursor movement (i.e. part #5).

Input file:

```
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
```

Interpreter interaction:

```
>>> e = Editor('text_files/lines.txt', 'output.txt')
>>> e.run()
     *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: i XYZ

     *
     12345678901234567890
```

```
*  1 XYZe 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: e

     *
     12345678901234567890
*  1 XYZe 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: q
>>>
```

Resulting output file:

```
XYZe 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
```

Below is another example. This example *does* involve cursor movement.
Input file:

```
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
```

Interpreter interaction:

```
>>> e = Editor('text_files/lines.txt', 'output.txt')
>>> e.run()
     *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
```

```
  10 Line 10
     12345678901234567890
Enter command: s

     *
     12345678901234567890
   1 Line 1
*  2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: d

      *
     12345678901234567890
   1 Line 1
*  2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: d

       *
     12345678901234567890
   1 Line 1
*  2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: i abracadabra

       *
     12345678901234567890
   1 Line 1
*  2 Liabracadabra
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: s

       *
     12345678901234567890
   1 Line 1
   2 Liabracadabra
*  3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
```

```
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: s

       *
     12345678901234567890
   1 Line 1
   2 Liabracadabra
   3 Line 3
*  4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: d

        *
     12345678901234567890
   1 Line 1
   2 Liabracadabra
   3 Line 3
*  4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: d

         *
     12345678901234567890
   1 Line 1
   2 Liabracadabra
   3 Line 3
*  4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: d

          *
     12345678901234567890
   1 Line 1
   2 Liabracadabra
   3 Line 3
*  4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: d

           *
     12345678901234567890
   1 Line 1
   2 Liabracadabra
   3 Line 3
*  4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
```

```
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: d

            *
     12345678901234567890
   1 Line 1
   2 Liabracadabra
   3 Line 3
*  4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: d

             *
     12345678901234567890
   1 Line 1
   2 Liabracadabra
   3 Line 3
*  4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: i blah

             *
     12345678901234567890
   1 Line 1
   2 Liabracadabra
   3 Line 3
*  4 Line 4  blah
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: e

             *
     12345678901234567890
   1 Line 1
   2 Liabracadabra
   3 Line 3
*  4 Line 4  blah
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: q
>>>
```

Resulting output file:

```
Line 1
Liabracadabra
Line 3
Line 4  blah
Line 5
Line 6
```

```
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
```

Below is an example where the insertion would extend past the end of the file. You do not need to print the error message, as the `run()` method already does this; your `insert()` method merely needs to return the correct boolean value.

```
...

            *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
   6 jklajs;lkfj
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: i akjsdlk;fjasklfjlksa;jfl;kj

ERROR: Invalid write.

            *
     12345678901234567890
*  1 abcdef
   2 ghi
   3 jklm
   4 jkj
   5 jklasj
   6 jklajs;lkfj
   7 qiowuioj
   8 lkjw;lkj
   9 qwklejklj;c
  10 jkaljsoiu
     12345678901234567890
Enter command: q
>>>
```

5.7 Part #7 - Edit History

In this part, you will modify the editor so that insertions can be undone/redone. Note that insertion is the only operation that this applies to; operations like cursor movements and saving cannot be undone/redone.

Below are some implementation tips and/or restrictions that apply to both parts #7.1 and #7.2 but that may not make sense until you read the directions for those parts.

- **As you can see in the `Editor` initializer as it was provided on Canvas, the `undo_history` and `redo_history` members are set to instances of `class Stack`. You are not allowed to change this. You must create a class used to represent an insertion operation. (If you want, you can have a class representing an undo-able insertion operation and a separate class representing a redo-able insertion operation.) This class / these classes should have enough information for you to undo or redo the insertion. Your stacks must only ever contain instances of this class / these classes. You are *not* allowed to store entire file buffers/contents in the stacks (i.e. do not store 2D lists in your stacks), as this would be unacceptably inefficient in terms of memory.**
- My own implementations of `undo()` and `redo()` are about 10 lines each.

5.7.1 Part #7.1 - Undoing an Insertion: `undo()`

When the undo key is entered, the `run()` method calls the `undo()` method. Implement the `undo()` method so that calling it completely undoes the last insertion that was done. Your editor must preserve an undo history that allows the user to undo as many operations as they wish, until there are no more operations to undo.

To be clear, there are only two ways in which an operation can be *added* to the undo history:

1. An insertion is done (i.e. part #6).
2. An insertion is redone (i.e. part #7.2).

The function should return `False` if there are no commands to undo, in which case the `run()` method will already know to print the appropriate message. Otherwise, the function should return `True`. Note that an infinitely long[3] undo history is supported.

Below are examples of how undoing an insertion should behave.

```
>>> e = Editor('text_files/lines.txt', 'ignore')
>>> e.run()
     *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: i abc

     *
     12345678901234567890
*  1 abce 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: u

     *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: u

ERROR: No operation to undo.

     *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: s
```

[3] Or, at least, as long as the amount of memory that Python will let you use.

```
     *
     12345678901234567890
   1 Line 1
*  2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: i abcd

     *
     12345678901234567890
   1 Line 1
*  2 abcd 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: i XYZ

     *
     12345678901234567890
   1 Line 1
*  2 XYZd 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: u

     *
     12345678901234567890
   1 Line 1
*  2 abcd 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: u

     *
     12345678901234567890
   1 Line 1
*  2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
```

```
Enter command: q
>>>
```

5.7.2 Part #7.2 - Redoing an Insertion: `redo()`

Implement the `redo()` method so that it redoes (re-does?) the last operation that was added to the redo history. The only reason that an operation should be added to the redo history is if an operation was undone; any undone insertion should be added to the redo history. *If the user does a new insertion (i.e. as in part #6), then the entire redo history should be erased. This is similar to the behavior of typical editors.*

The function should return `False` if there are no commands to redo, in which case the `run()` method will already know to print the appropriate message. Otherwise, the function should return `True`. An infinitely long redo history is supported.

Below are examples of how redoing an insertion should behave.

```
>>> e = Editor('text_files/lines.txt', 'ignore')
>>> e.run()
     *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: i hi there friend

     *
     12345678901234567890
*  1 hi there friend
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: u

     *
     12345678901234567890
*  1 Line 1
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: r

     *
     12345678901234567890
*  1 hi there friend
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
```

```
  10 Line 10
     12345678901234567890
Enter command: r

ERROR: No operation to redo.

     *
     12345678901234567890
*  1 hi there friend
   2 Line 2
   3 Line 3
   4 Line 4
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
     12345678901234567890
Enter command: u

[... skipping many cursor movements ...]

       *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
* 12 Line 12
  13
  14
     12345678901234567890
Enter command: d

        *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
* 12 Line 12
  13
  14
     12345678901234567890
Enter command: i blah blah

        *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
  10 Line 10
  11 Line 11
* 12 Linblah blah
  13
  14
     12345678901234567890
Enter command: w

        *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
```

```
   9 Line 9
  10 Line 10
* 11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: w

        *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
* 10 Line 10
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: d

         *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
* 10 Line 10
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: i haha

         *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
   9 Line 9
* 10 Linehaha
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: w

         *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
*  9 Line 9
  10 Linehaha
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: i abcd

         *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
```

```
   8 Line 8
*  9 Lineabcd
  10 Linehaha
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: i XYZ

         *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
*  9 LineXYZd
  10 Linehaha
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: u

         *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
*  9 Lineabcd
  10 Linehaha
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: u

         *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
*  9 Line 9
  10 Linehaha
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: r

         *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
*  9 Lineabcd
  10 Linehaha
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: r

         *
     12345678901234567890
   5 Line 5
   6 Line 6
```

```
   7 Line 7
   8 Line 8
*  9 LineXYZd
  10 Linehaha
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: u

         *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
*  9 Lineabcd
  10 Linehaha
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: i :(

         *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
*  9 Line:(cd
  10 Linehaha
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: r

ERROR: No operation to redo.

         *
     12345678901234567890
   5 Line 5
   6 Line 6
   7 Line 7
   8 Line 8
*  9 Line:(cd
  10 Linehaha
  11 Line 11
  12 Linblah blah
  13
  14
     12345678901234567890
Enter command: q
>>>
```

UCDAVIS
COMPUTER SCIENCE