Console-Based Editor - S3 - 2026

```
CSC1002>python3 editor.py
>iA simple, basic editor by CSC1002
A simple, basic editor by CSC1002
>w
A simple, basic editor by CSC1002
>w
A simple, basic editor by CSC1002
>$
A simple, basic editor by CSC1002
>b
A simple, basic editor by CSC1002
>b
A simple, basic editor by CSC1002
>^
A simple, basic editor by CSC1002
>.
A simple, basic editor by CSC1002
>w
A simple, basic editor by CSC1002
>.
A simple, basic editor by CSC1002
>
```

OVERVIEW

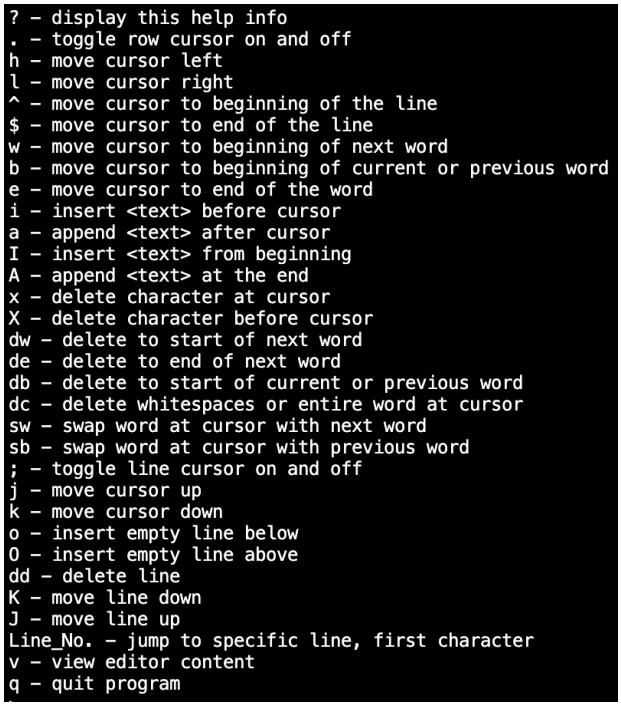
In this assignment, you are going to design and develop a simple, basic console-based editor. Unlike the modern, advanced editor which provides a sophisticated editing environment, utilizing the high-resolution of the graphical screen together with the mouse and the keyboard to position and adjust any text and figures displayed on the screen, giving us the WYSIWYG (What-You-See-Is-What-You-Get) experience.

In the early days, lacking access to a rich graphical display and mouse, the functionality of editors was limited, providing only a much simpler user interface, usually console-based. Editing was carried out based on simple text commands entered via the keyboard, commands such as inserting (i) and appending (a) a text string, positioning the editor cursor one character position to the left (h), one character position to the right (l), one-word position forwards (w), one-word position backwards (b), and so on.

```
A simple, basic editor by CSC1002
>h
A simple, basic editor by CSC1002
>h
A simple, basic editor by CSC1002
>h
A simple, basic editor by CSC1002
>h
A simple, basic editor by CSC1002
>l
A simple, basic editor by CSC1002
>l
A simple, basic editor by CSC1002
>$
A simple, basic editor by CSC1002
>^
A simple, basic editor by CSC1002
>$
A simple, basic editor by CSC1002
>a,Kinley/SSE
A simple, basic editor by CSC1002,Kinley/SSE
```

SCOPE

1. Complete all the following editor commands:



NOTE: Refer to the section “Specific Spec” for detailed information on specific requirements for certain commands. The description above conveys the command's general intent at a high level.

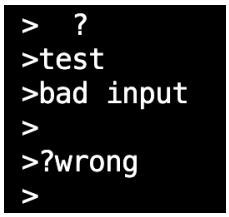
2. Case-sensitive commands - all editor commands are case-sensitive, for example, the capital letter ‘A’ does not equal the lowercase letter ‘a’.

3. Command types - most commands are single letters such as ?, $, x, ^, …etc, while some are two-letter commands. Most commands do not require extra input, while a few do, such as insert (i) and append (a).

4. Command prompt (>) - The prompt is a single character string ‘>’, without any leading or trailing spaces, and the input function - input(‘>’) - is used to display the prompt string. See the screenshots on the first page.

5. Command Syntax - Command[Text], where “Command” is one of the commands shown in step 1, “Text” applies only to commands requiring extra input such as insert (i) and append (a). Any commands whose description includes a substring enclosed in “<>” brackets require extra input “Text”.

6. Command Parsing - when prompted (>) users enter one command at a time including any additional required input and then press the return key to continue. Parse each command string according to “Command Syntax” to ensure that the input string matches EXACTLY one of the commands from step 1, including the extra input “Text” if required. When invalid input is entered, simply display another prompt as illustrated in the following screenshot.



The following examples show the user input enclosed by a pair of double quotes; they represent the exact user input, including any leading or trailing whitespace. This distinction is important for determining whether an input is valid. For instance, "i" is not valid because the insert command requires an additional string to specify what should be inserted. By contrast, "i " is valid, since it inserts a single empty space. Always pay close attention to whitespace in the examples, as it can change the meaning of the command.

   a. Examples of valid user input:
      i. “$”
      ii. “^”
      iii. “h”
      iv. “i ”
      v. “a “
      vi. “a hello world”
      vii. “i hello world “
   b. Examples of invalid user input:
      i. “ $”
      ii. “ ?”
      iii. “? “
      iv. “ a hello world”
      v. “i”
      vi. “a”

7. Command Execution - the editor repeatedly prompts the user to enter a command, parses the user input according to the "Command Syntax", and then performs the operation described as follows:

    a. if the user input is valid (with the required additional input, if any), executes the command, outputs the recent editor content (even there are no changes to the content) on the console (except for commands '?' and 'q', see Note follows) and then displays another prompt ('>') immediately after the editor content.

    b. If the user input is invalid (unknown command, missing required extra input, …etc.), do not display the editor content and do not display any error message, instead display another prompt ('>') immediately to ask the user to enter the command again.

    Note: when the help command (?) is entered, display the help menu as shown in step 1 followed immediately by another prompt ">"; when the quit command ('q') is entered, end the program immediately without any additional output.

NOTE:

- Keep your entire source code in ONE SINGLE file.
- Use only Python modules as specified in the "Permitted Modules" section.
- In your design stick ONLY to functions, in other words, no class objects of your own.
    - Furthermore, the lines of code containing the sub-function(s) defined within another function will be counted as part of the parent function.
    - NOTE: Failure to adhere to the instructions outlined in the assignment handout will result in maximum 50% reduction in the coding style score.

SPECIFIC SPEC

Note 1: Please read each passage carefully, and review the accompanying examples (if provided) to guide your understanding. They are intended to clarify expectations. You should examine these examples thoroughly.

Note 2: This section DOES NOT cover every command. It focus only on those commands that require further clarification.

1. Editor (content and prompt) - In the editor program, the content and the prompt are handled separately. The editor displays its content using the standard print() statement. When the program first starts, the editor contains a single empty line. Each time the editor refreshes, it passes the current content to print(), which by default appends an end-of-line and line-feed after the text.

   By contrast, the input() function is used to display the prompt and accept user commands. The prompt is a single character ">", shown exactly as input(">"), with no spaces before or after. The prompt does not represent the editor's content; it simply signals that the editor is ready for another command. Likewise, the print() statement does not include the prompt string.

   For example, if the current content of the editor is "hello world", use print("hello world") to display the content, immediately followed by input(">") to display the prompt.

```
# correct
print("hello world")
input(">")
```

```
hello world
>
```

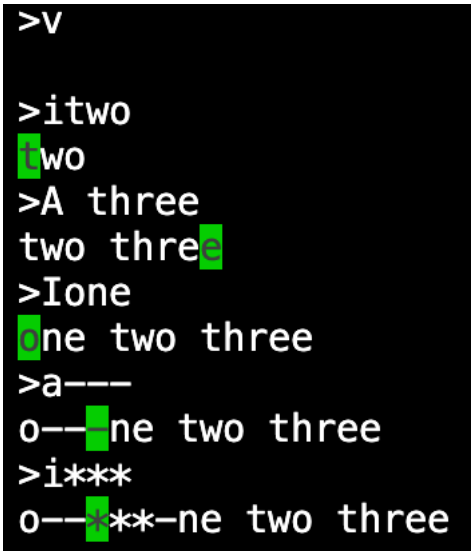
2. Row cursor - it's used to show where the cursor is on the current row if not empty. In other words, the cursor will appear only on printable characters including space, not on empty line. The row cursor, if enabled, is shown by wrapping the character under the cursor with a pair of escape character strings such as "\033[42m" and "\033[0m". For example, given a string "hello world", to show the green cursor at the position of the letter 'e', this is the string to print: "h" + "\033[42m" + "e" + "\033[0m" + "llo world". When the row cursor is not enabled, do not include any escape character strings in the print statement. Note: when the program first starts, the row cursor is enabled by default.

3. Word boundary - a word is defined as a sequence of consecutive, printable characters, including other non-alphabetic letters, but not white spaces. In other words, any group of characters without spaces is considered a single word, even if it includes punctuation characters like '?', '-', '=', and so on.

```
>.
 one-one   two,two  three**three--
>b
 one-one   two,two  three**three---
>b
 one-one   two,two  three**three---
>b
 one-one   two,two  three**three---
>e
 one-one   two,two  three**three---
>e
 one-one   two,two  three**three---
>e
 one-one   two,two  three**three--
```

Text Insertion Commands

4. Insert<Text> (i) - inserts the given string <Text> immediately to the left of the cursor, or into an empty line if the line has no content. After insertion, the cursor moves to the beginning of the newly inserted <Text> string.
5. Insert<Text> (I) - inserts the given string <Text> at the beginning of the current line, regardless of the cursor's position. After insertion, the cursor moves to the beginning of the line.
6. Append<Text> (a) - appends the given string <Text> immediately to the right of the cursor, or into an empty line if the line has no content. After appending, the cursor moves to the end of the newly inserted <Text> string.
7. Append<Text> (A) - appends the given string <Text> at the end of the current line, regardless of the cursor's position. After appending, the cursor moves to the end of the line.



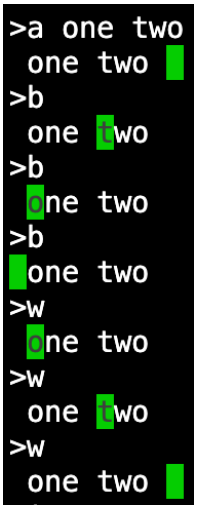
Cursor Motion Commands

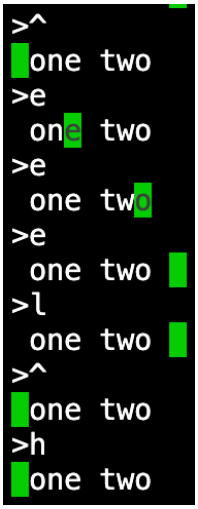
8. Next Word (w) - moves the cursor forward to the beginning of the next word; if the cursor is already positioned within the last word or beyond the last word, set the cursor to the end of the line.

9. Previous Word (b) - moves the cursor backward to the beginning of the current or previous word; if the cursor is already positioned to the left or at the beginning of the first word, set the cursor to the beginning of the line.

10. End Word (e) - moves the cursor forward to the end of the current or next word; if the cursor is at the end of a word, set the cursor forward to the end of the next word (if any). If the cursor is already positioned beyond or at end of the last word, set the cursor to the end of the line.

11. Left (h), Right (l) - moves the cursor one position to the left or to the right, and if the cursor is already at the far left or far right position, leaves the cursor where it is.

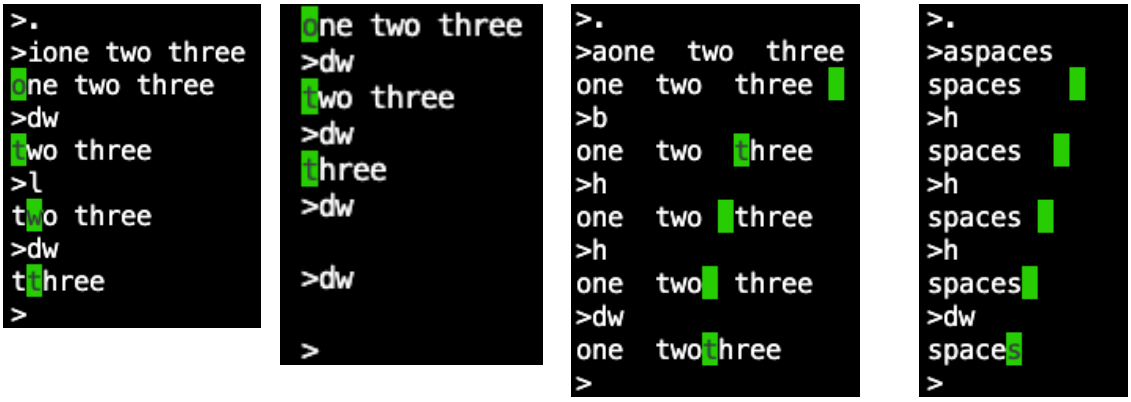




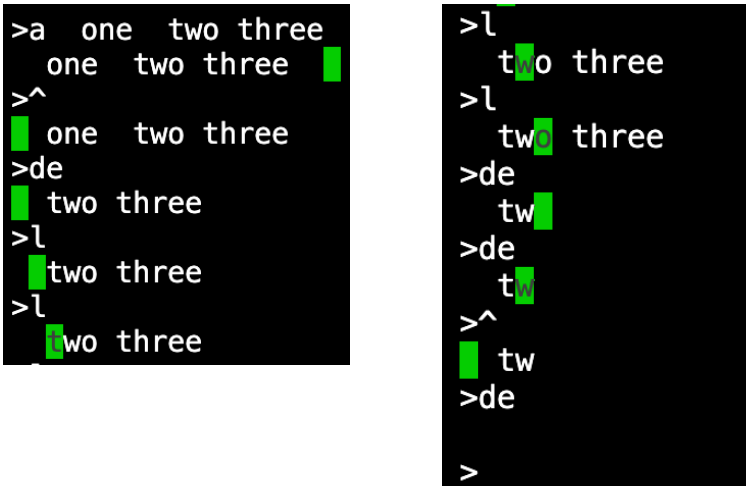
Delete Commands

Note: The following are some of the delete commands with further clarification. In general, when all characters are deleted, the editor buffer is reset to an empty line. In addition, carefully review the accompanying examples, if provided.

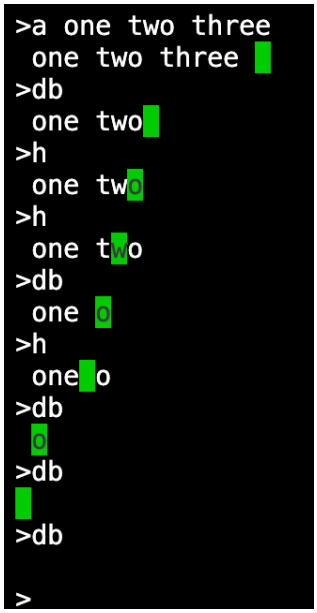
12. Delete next word (dw) - deletes all characters from the cursor position forward, up to but excluding the position defined by the motion command "w" if the position is the beginning of a word. In other words, "dw" removes all characters in between, including any whitespace, and if there is no next word, it deletes through to the end of the line. After the operation, the cursor is positioned at the start of the next word, if one exists, or at the end of the line otherwise.



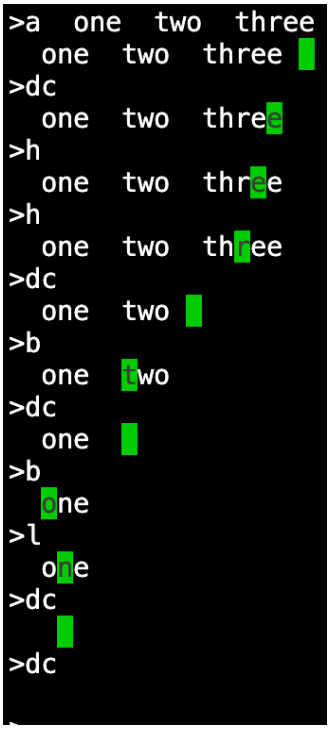
13. Delete end word (de) - deletes all characters from the cursor position forward, up to and including the position defined by the motion command "e". In other words, "de" removes all characters starting from wherever the cursor is (whether at the start, middle, or end of a word, or even between words) through the last character of the current word or the next word. If no further word exists, the deletion extends to the end of the line. After the operation, the cursor remains at its position unless the deletion reaches the end of the line, in which case it shifts one character backward to the last remaining character.



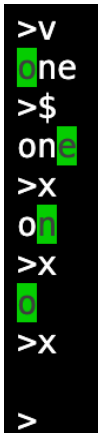
14. Delete previous word (db) - deletes all characters from the cursor position backward, up to and including the position defined by the motion command b. In other words, db removes text from wherever the cursor is (whether at the start, middle, or end of a word, or between words) back through to the beginning of the current or previous word. If there is no previous word, the deletion extends to the start of the line. After the operation, the cursor is positioned at the beginning of the word that remains, or at the start of the line if no word precedes it.



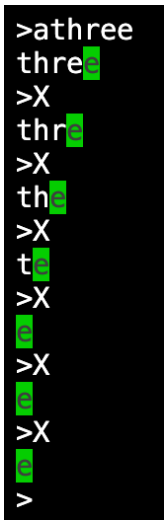
15. Delete at cursor (dc) - deletes either the entire word under the cursor (whether at the start, middle, or end of a word), or all spaces surrounding and under the cursor if the cursor is not positioned within a word. After the operation, the cursor remains at its original position, if feasible, or adjusts accordingly to the nearest valid position.



16. Delete Character(the lowercase letter x) - deletes the character at the cursor position.

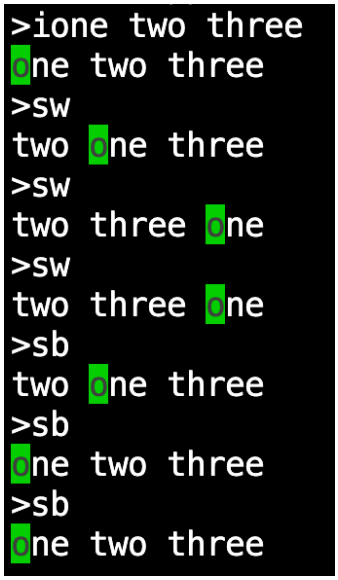


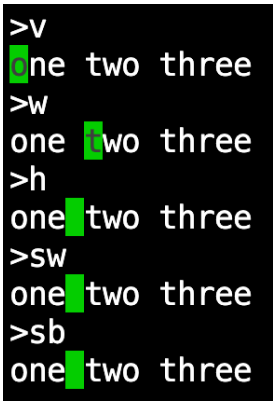
17. Delete Character(the uppercase letter X) - deletes the character immediately before the cursor position.



Swap Word Commands

18. Swap next word (sw) - swap the word under the cursor, including the cursor position, with the next word, if any. Take no action if the cursor is not within a word, whether at the start, middle, or the end of the word.
19. Swap previous word (sb) - swap the word under the cursor, including the cursor position, with the previous word, if any. Take no action if the cursor is not within a word, whether at the start, middle, or the end of the word.





Multi-Line Commands

Note: The following clarifications apply to some of the commands in the multi-line version of the editor. In general, when all lines are deleted, the editor content is reset to an empty line. In addition, carefully review the accompanying examples, if provided

20. Line Cursor (;) - Under the multi-line version of the editor, a second symbol is introduced to mark which line is currently active. The editor prefixes the active line with an asterisk (*). To preserve alignment, all other lines are shifted one space forward so that their text lines up neatly with the active line despite the extra cursor symbol.

    Initially, the line cursor is disabled. To make it visible, you must explicitly enable it using the line cursor command. Once enabled, the active line is always shown with *, making it easy to identify which line is in focus for editing operations.

21. Row Cursor Motion (j, k) - under the multi-edition version of the editor, it moves the row cursor vertically up or down relative to the current line. The lowercase “j” positions the row cursor to the line above, while “k” to the line below. If the target line is shorter than the current row cursor position, the row cursor automatically shifts to the end of the target line.

    When the cursor is already on the first line and j is issued, or on the last line and k is issued, the editor does not wrap around from top to bottom or vice versa. Instead, the line cursor remains on the current line, and the row cursor stays exactly where it is.

22. Insert Empty Line (o,O) - inserts a new empty line below or above the current line and the line cursor automatically moves to the newly inserted line, whether the line cursor is enabled or disabled. The lowercase “o” inserts an empty line below the current line, while the uppercase “O” above the current line.

```
>v

>;
*
>ione two three
*one two three
>o
 one two three
*
>j
*one two three

>O
*
 one two three

>
```

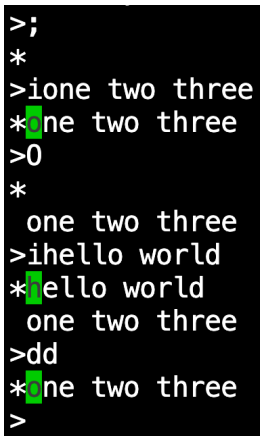
```
>ihello
hello
>;
*hello
>o
 hello
*
>ihello world
 hello
*hello world
>$
 hello
*hello world
>j
*hello
 hello world
```

```
>aone
one
>o
one

>aone two
one
one two
>o
one
one two

>aone two three
one
one two
one two three
```

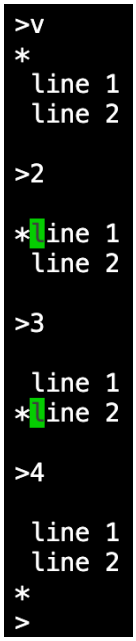
```
>;
*
>o

*
>k

*
>j
*

>j
*

>
```

23. Delete Line (dd) - deletes the entire current line, if the deleted line is the only line in the editor, the editor content becomes an empty line.

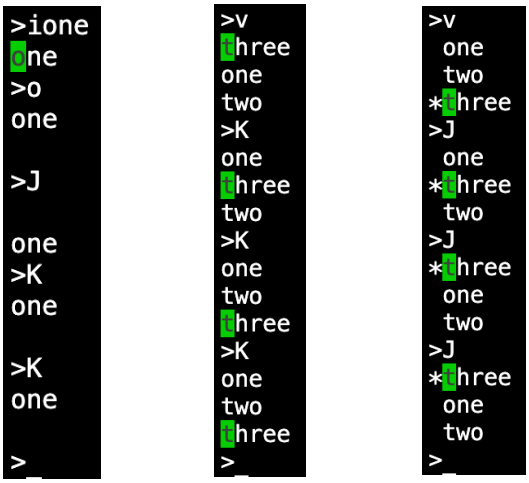
    Under the multi-line version, once the command is executed, the line is removed from the editor, and all lines below it (if any) shifts upward. The line that was previously below the deleted line take its place and becomes the active line. If the deleted line is at the bottom of the editor, the line cursor shifts upward one position, provided there is another line above. The row cursor position is preserved if possible; otherwise, it is adjusted to the end of the new active line.



24. Line_No. - jumps to a specific line and sets the row cursor to the first character of that line (if any). The line number (Line_No.) must be a positive integer greater than 0, with 1 referring to the top line. If the line number exceeds the total number of lines in the current editor, the line cursor is set to the last line.



25. Move Line (J, K) - repositions the current line either upward or downward by one line at a time. When moving a line upward (J), the current line trades places with the line immediately above it; when moving downward (K), it trades places with the line immediately below. After the operation, the line cursor continues to follow the moved line, ensuring that the same line remains active even though its position in the editor has changed. The row cursor position remains unchanged.



ASSUMPTIONS

1. The goal of this assignment is to illustrate the benefits of "Problem Decomposition", "Clean Coding" and "Refactoring", all together achieving high code readability to ease logic expansion and keep high maintainability, therefore, it's not aimed at designing a complex, general-purpose editor for handling large editing content.
2. It's assumed that the length of each line is kept within a reasonable length so that each line can be stored directly using the standard Python 'str' type. If editor supports multiple lines, the number of lines is also kept within a reasonable number so that all lines can be kept in one standard Python list and the lines can be efficiently updated using the standard list and str operations such as append, insert, slicing, cloning, …etc.
3. It is assumed that the user will not input a command that consumes excessive memory and leads to a buffer overflow (also called memory overflow or overrun) at runtime, such as inserting a very long string like "ihello …….. world." In other words, all test cases executed against your program will be based on the commands from step 1 (Scope) with a short "Text" string.
4. Each test case is designed to evaluate the functionality and correctness of your program, rather than its speed, performance and memory usage. Each test case consists of multiple editing commands with short "Text".
5. The text editor is required to handle only regular English characters, thus additional unicode support is unnecessary.

SKILLS

In this assignment, you will be trained on the use of the followings:

- Refactoring - logic reuse or simplification based on the existing logic.
- Variable scope: global, local and function parameters.
- Coding Styles (naming convention, meaningful names, comments, doc_string, …etc)
- Problem Decomposition, Clean Code, Top-Down Design
- Functions (with parameters and return) for program structure and logic decomposition
- Standard objects (strings, numbers & lists)
- Variable Scope

PERMITTED MODULES

Only the following Python module(s) is allowed to be used:

- re (regular expression)

DELIVERABLES

Program source code (AX_School_StudentID.py), where X is the assignment number (1,2,3,..) and School is SSE, SDS, SME, HSS, FE, LHS, MED and StudentID is your 9-digit student ID.

For instance, a student from SME with student ID "119010001" names the Python file of first assignment as follows:

- A1_SME_119010001.py

Ensure that your source file is saved in standard, regular UTF-8 encoding format. On the status bar of Visual Studio Code, you can view the current encoding format as follows:

On an occasion, the encoding scheme is set to UTF-8 with BOM as follows:

The presence of the Byte Order Mark (BOM) could be due to copying from websites, older version of editor, file conversion from other sources, default encoding setting, and so on.

Confirm the encoding scheme is UTF-8 and the file name is correct, then submit the plain program file to the corresponding assignment folder. A deduction of 5% will be penalized if the file is incorrectly named or in wrong encoding format.

TIPS & HINTS

- Apply problem decomposition, Clean Code and Refactoring as illustrated during classes.
- Beware of variable scope as you might keep a few variables as global such as current editor content, cursor position, cursor state, and so on.
- Refer to Python website for program styles and naming conventions (PEP 8)

MARKING CRITERIA

- Coding Styles – overall program structure including layout, comments, white spaces, naming convention, variables, indentation, functions with appropriate parameters and return.
- Program Correctness – whether or the program works 100% as per Scope.
- User Interaction – how informative and accurate information is exchanged between your program and the player.
- Readability counts – programs that are well structured and easy to follow using functions to break down complex problems into smaller cleaner generalized functions are preferred over a function embracing a complex logic with many nested conditions and branches! In other words, a design with a clean architecture and high readability is the predilection for the course objectives over efficiency. The logic in each function should be kept simple and short, and it should be designed to perform a single task and be generalized with parameters as needed.
- KISS approach – Keep It Simple and Straightforward.
- Balance approach – you are not required to come up with a very optimized solution. However, take a balance between readability and efficiency with good use of program constructs.

ITEMS	PERCENTAGE	REMARKS
CODING STYLE	40%	THE CODING STYLE SCORE WILL BE ADJUSTED PROPORTIONALLY TO FUNCTIONALITY. SEE NOTE BELOW.
FUNCTIONALITY	60%	REFER TO SCOPE

NOTE:

Functionality - determined by the percentage of test cases passed.

Coding Style - determined by linter metrics (e.g., code complexity, number of statements per function, branching, nesting levels, and so on) and adjusted proportionally to functionality. For examples:

- If a program passes all test cases, the student receives the full coding style score.
- If a program passes 90% of test cases, the student receives 90% of the coding style score.
- If a program fails all test cases, the student receives zero overall score, and so on.

Coding style is recognized only when the program demonstrates functional correctness, and full credit is reserved for programs that meet all requirements, while the deduction is a reflection of incomplete functionality.

CODING STYLE - SAMPLE METRICS

Coding style will be evaluated based on common code smells such as:

- Long functions
- Too many parameters
- Excessive branching
- Deeply nested logic
- Other readability and maintainability issues

A table of metrics is provided to illustrate how these code smells are assessed. This list is not exhaustive. To the best of your effort, you are expected to apply the principles and best practices discussed in lectures and demonstrated during class sessions, reflecting your understanding of clean coding principles.

Code Smells/Clean Coding	Metric
Long Function	Max. 9 LOC
Parameters	Max. 4
Branches	Max. 6
Variables+Parameters per function	Max. 8
Nested Level	Max. 2
Long Statement	100 characters
Commenting (function, module)	Docstring
Naming Style (variable, parameter, function)	snake_case
Meaningful names	Min. 3 characters (local, parameters) Min. 5 characters (global)
….	….

LOC - number of logical statements (excluding Docstring, comments, whitespaces, blank lines)

DEMO TERMINALS

Instead of handling out a few sample test cases, two virtual machines are available so that you can connect to them via SSH (Secure-Shell). They are provided to let you explore and understand more about the behavior of the editor program interactively. Once connected, the editor program will be launched automatically.

Intentions:

- Explore the program's behavior interactively.
- Test your understanding of the assignment requirements.
- Experiment freely with different inputs and scenarios.
- Develop your own test cases based on what you observe.

Rules and limits (due to limited resources):

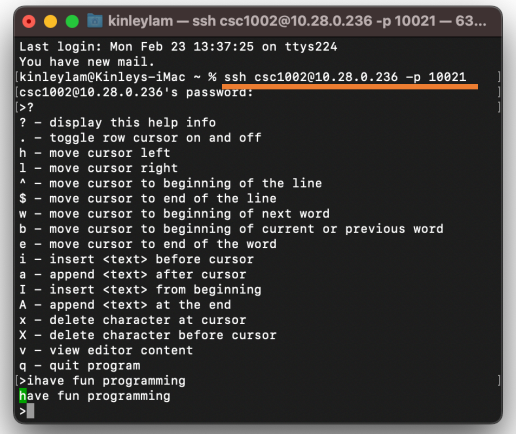
- Each machine supports a maximum of 10 concurrent sessions.
- Idle sessions are closed after 2 minutes.
- Do not wait until the last minute to connect - you may be locked out if the machines are full or your session times out.
- Always follow the assignment handout first; use the demo terminals if you need to explore further.
- Any misuse, unauthorized actions, or attempts to disrupt the system will result in disciplinary consequences.

SSH Info:

- IP address: 10.28.0.236
- User Account: csc1002
- Port: 10021 or 10022
- Password: csc1002

Examples:

```
ssh csc1002@10.28.0.236 -p 10021
ssh csc1002@10.28.0.236 -p 10022
```



DUE DATE

Submit assignment by end of April 24, 2026