# Wordle

Assignment 1
Semester 1, 2022
CSSE1001/CSSE7030

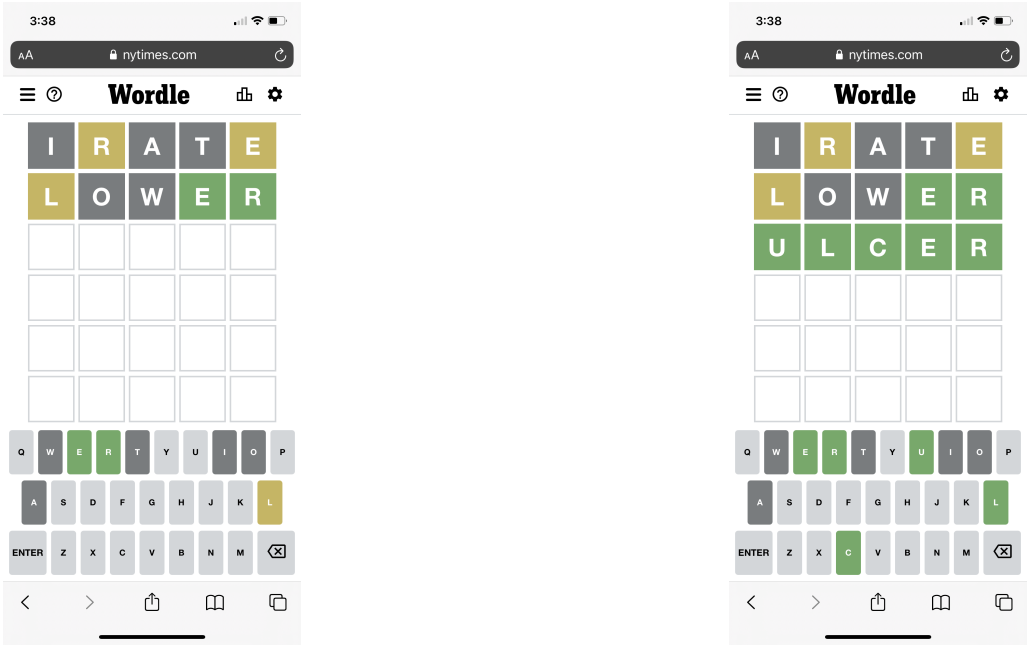
Due date: 1 April 2022, 16:00 GMT+10

## 1 Introduction

The word-guessing game Wordle[1] has recently gained huge popularity, and despite being fairly simple, was recently sold to the New York Times for an undisclosed price "in the low seven figures". Once a day, players have up to six attempts to guess a word. After each guess, players receive the following feedback:

- Letters from the guess that are in the answer **and** in the correct position are coloured green;
- Letters from the guess that are in the answer but **not** in the correct position are coloured yellow; and
- Letters from the guess that are not in the answer are coloured grey.

Figure 1 demonstrates a partially and fully completed game of Wordle, where the answer is 'ulcer'.




(a) A partially completed game.

(b) The game is solved in three guesses.

Figure 1: Guess letters are coloured according to the rules, and keyboard letters are coloured according to current known information.

[1] `https://www.powerlanguage.co.uk/wordle/`

In this assignment you will implement a modified text-based version of Wordle. CSSE7030 students will also implement a simple solver to assist the user in making guesses during the game. You are required to implement a number of functions as per Section 5 of this document.

# 2 Getting Started

Download `a1.zip` from Blackboard — this archive contains the necessary files to start this assignment. Once extracted, the `a1.zip` archive will provide the following files:

`a1.py` *This is the only file you will submit* and is where you write your code. *Do not* make changes to any other files.

`a1_support.py` *Do not modify or submit this file*, it contains functions to help you implement your tasks. You do not need to read or understand the code itself, but you should read the docstrings in order to get an idea of how the functions work. This file also contains some constants that will be useful in your implementation. In addition to these, you are encouraged to create your own constants in `a1.py` where possible.

`vocab.txt` This is the file containing the entire vocabulary of valid guesses, including potential answers.

`answers.txt` This is a reduced version of `vocab.txt`, from which answers will be randomly chosen for each round. Separating answers from the greater vocabulary ensures all answers are commonly known words, while allowing players to guess uncommon words.

`examples/` This a folder containing example text file outputs from a working assignment 1 solution. The purpose of these files is to help you analyse the formatting requirements (e.g. amount of whitespace) of the output. Note that while you should be able to replicate the CSSE1001 example exactly, those completing the CSSE7030 task may find their solution produces different guesses to the CSSE7030 example while still being correct. However, all other aspects of your ouptut should match the file (e.g. formatting).

***NOTE:*** You are not permitted to add any import statements to `a1.py`. Doing so will result in a deduction of up to 100% of your mark.

# 3 Gameplay

This section provides an overview of gameplay. For exact prompts and outputs, see Figure 2, and Sections 5 and 6. When your program is run, the user should be prompted for their first guess. When prompted for an input, users may do one of four (or five for CSSE7030 students) things:

1. Enter a guess. Guesses must be 6 letters long and exist in the known vocabulary. After a valid guess is entered, all previous guesses should be displayed with their information, before the user is prompted for their next guess.

2. Enter either 'k' or 'K'. After displaying the keyboard information (see Section 5.8), the user should be re-prompted for their guess.

3. Enter either 'q' or 'Q'. The user is opting to quit the game, and the program should terminate.

4. Enter either 'h' or 'H'. The help message "Ah, you need help? Unfortunate." should be printed, and the user should be re-prompted for their guess.

5. *CSSE7030 students only*: Enter either 'a' or 'A' to have the solver make the next guess. See Section 6 for more details.

If the user does not enter input that matches any of the above cases, it is assumed they have entered an invalid guess. In this case, the user should be informed of the way in which their guess is invalid (see Fig. 2 for the exact text you must match), before being reprompted for their guess.

The user continues until either:

- They *win* the round by guessing the word correctly within 6 guesses.
- They *lose* the round by entering 6 consecutive non-winning guesses.
- They quit the game by entering 'q' or 'Q'.

At the end of a round, the user is informed of their outcome, along with their stats, before being prompted for whether they would like to play again. At this prompt, the user can either:

1. Enter either 'y' or 'Y' to play another round, in which case another word is chosen from the answers and another round commences with this new answer. You must not select the same word twice.
2. Enter *anything else*, in which case the game ends and the program terminates.

Figure 2 shows example gameplay for a fully functioning CSSE1001 Assignment 1. Here, the user opts to play two rounds, and then quit. If they had lost a round, the lose text should read "You lose! The answer was: `{answer}`" with the correct answer substituted for `{answer}`. If you have correctly handled the random selection of answer words, you should have the same first two answer words, and be able to replicate this output exactly. You should also test your program with other inputs.

# 4 Valid and invalid assumptions

In implementing your solution, you *can* assume that:

- All words (answers and full vocabulary) have exactly six letters. Any guess that does not contain exactly six letters is invalid.
- All *answers* will consist of six *unique* letters. No duplicate letters appear in answers.
- Your program will not be tested with guesses containing uppercase letters.
- A guess is valid if it is present in `vocab.txt`.
- Your program will not be tested for more rounds than there are available answers.

You *must not* assume that:

- All valid guesses may only contain unique letters. While answers will not contain duplicate letters, a valid guess could include duplicate letters.
- Guesses will only be valid 6 letter words. Your program will be tested with invalid words, such as those containing special characters (e.g. non-letter characters) and words of incorrect length. These situations must be handled as specified in Section 3.
- The `vocab.txt` and `answers.txt` files used in grading your assignment will be identical to the ones given to you for development purposes. We may choose to mark your assignment using new words in the contents of these files, though the file names and structure will stay consistent.

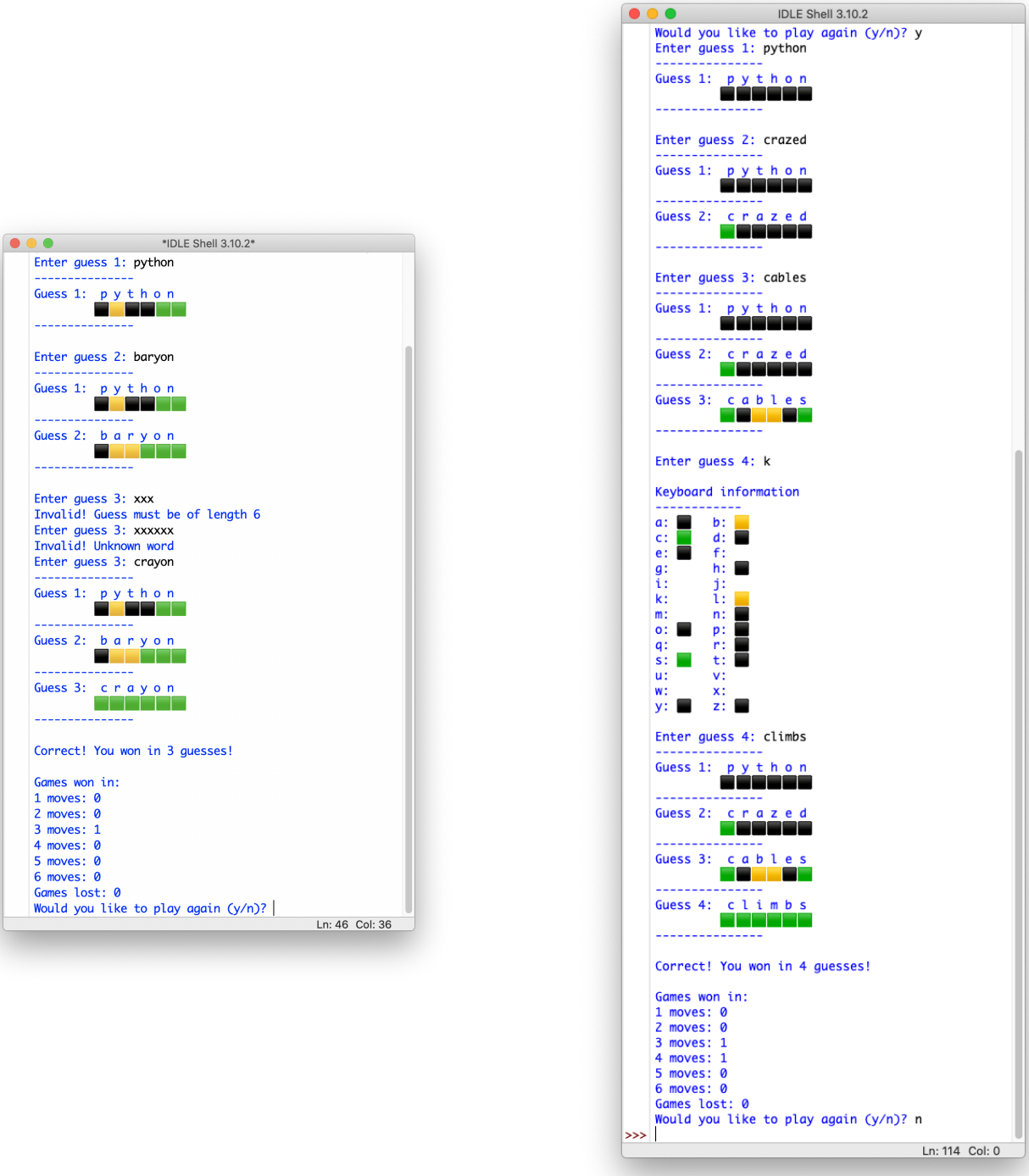



Figure 2: Basic gameplay demonstration. Blue text is program output. Black text is user input.

# 5 Implementation

This section outlines the functions you are *required* to write. You are awarded marks for the number of tests passed by your functions when they are tested *independently*. Thus an incomplete assignment with *some* working functions may well be awarded more marks than a complete assignment with faulty functions. Your program must operate *exactly* as specified. In particular, your program's output must match *exactly* with the expected output. Your program will be marked automatically so minor differences in output (such as whitespace or casing) *will* cause tests to fail resulting in a *zero mark* for that test.

Each function is accompanied with some examples for usage to help you *start* your own testing. You should also test your functions with other values to ensure they operate according to the descriptions.

Note: If you are using an IDE other than IDLE, the squares may render as a different size. Your solution is correct if and only if it passes the tests of gradescope.

Also note: IDLE 3.9 will not run a solution to this assignment. If you are using IDLE to develop you should use IDLE 3.10.

## 5.1 `has_won(guess: str, answer: str) -> bool`

Returns True if the guess `guess` matches `answer` exactly. Otherwise, returns False.
Precondition: `len(guess) == 6 and len(answer) == 6`

### 5.1.1 Example usage

```
>>> has_won('python', 'python')
True
>>> has_won('candle', 'python')
False
>>> has_won('python', 'candle')
False
```

## 5.2 `has_lost(guess_number: int) -> bool`

Returns True if the number of guesses that have occurred, `guess_number`, is equal to or greater than the maximum number of allowed guesses (in our case, six). Otherwise, returns False.

### 5.2.1 Example usage

```
>>> has_lost(1)
False
>>> has_lost(6)
True
>>> has_lost(5)
False
```

## 5.3 `remove_word(words: tuple[str, ...], word: str) -> tuple[str, ...]`

Returns a *copy* of `words` with `word` removed, assuming that `words` contains `word` exactly once.

### 5.3.1 Example usage

```
>>> words = ('python', 'apples', 'candle')
>>> new_words = remove_word(words, 'candle')
>>> words
('python', 'apples', 'candle')
>>> new_words
('python', 'apples')
```

## 5.4 `prompt_user(guess_number: int, words: tuple[str, ...]) -> str`

Prompts the user for the next guess, reprompting until either a valid guess is entered, or a selection for help, keyboard, or quit is made. Returns the first valid guess or request for help, keyboard, or quit. Note that the processing of the guess once it has been entered does not need to be handled in this function; it will be handled in functions defined later in this section. The returned string must be lowercase.

### 5.4.1 Example usage

```
>>> vocab = load_words("vocab.txt")
>>> guess = prompt_user(4, vocab)
    Enter guess 4: x
    Invalid! Guess must be of length 6
    Enter guess 4: xxxxxx
    Invalid! Unknown word
    Enter guess 4: debunk
>>> guess
    'debunk'
```

## 5.5 `process_guess(guess: str, answer: str) -> str`

Returns a modified representation of guess, in which each letter is replaced by:

- A green square if that letter occurs in the same position in answer;
- A yellow square if that letter occurs in a different position in answer; or
- A black square if that letter does not occur in answer.

While answers must contain 6 unique letters, guesses may contain duplicate letters. If duplicate letters exist in the guess, only *one* can have a non-black square. If the letter doesn't exist in the answer, all occurrences of said letter in guess are given black squares. If the letter does exist in answer, the non-black square is allocated as follows:

1. If one of the occurrences is in the correct position, it receives a green square and all other occurrences receive a black square.
2. Otherwise, if no occurrences are in the correct position, the first occurrence of the letter in guess receives a yellow square and all other occurrences receive a black square.

Precondition: `len(guess) == 6 and len(answer) == 6`
See `a1_support.py` for constants containing the required square characters.

### 5.5.1 Example usage

```
>>> process_guess('debunk', 'dances')
    '🟩🟨⬛⬛🟨⬛'
>>> process_guess('candle', 'dances')
    '🟨🟩🟩🟨⬛🟨'
>>> # Here, the second r gets the non-black square because it's in the right position
>>> process_guess('arrows', 'strand')
    '🟨⬛🟩⬛⬛🟨'
>>> # Here, the first r gets the non-black square because no r's are in the right position but
>>> # r exists in the answer
>>> process_guess('arrows', 'answer')
    '🟩🟨⬛⬛🟨🟨'
```

## 5.6 `update_history( history: tuple[tuple[str, str], ...], guess: str, answer: str ) -> tuple[tuple[str, str], ...]`

Returns a *copy* of `history` updated to include the latest `guess` and its processed form. See Section 5.7 for example usage of this function. `history` is a tuple where each element is a tuple of `(guess, processed_guess)`.

## 5.7 `print_history(history: tuple[tuple[str, str], ...]) -> None`

Prints the guess history in a user-friendly way.

### 5.7.1 Example usage

```
>>> answer = 'dances'
>>> history = ()
>>> history = update_history(history, 'python', answer)
>>> history
(('python', '⬛⬛⬛⬛⬛🟨'),)
>>> print_history(history)
---------------
Guess 1:  p y t h o n
          ⬛⬛⬛⬛⬛🟨
---------------

>>> history = update_history(history, 'debunk', answer)
>>> history
(('python', '⬛⬛⬛⬛⬛🟨'), ('debunk', '🟩🟨⬛⬛🟨⬛'))
>>> print_history(history)
---------------
Guess 1:  p y t h o n
          ⬛⬛⬛⬛⬛🟨
---------------
Guess 2:  d e b u n k
          🟩🟨⬛⬛🟨⬛
---------------

>>>
```

## 5.8 `print_keyboard(history: tuple[tuple[str, str], ...]) -> None`

Prints the keyboard in a user-friendly way with the information currently known about each letter. Note that the two columns are tab-separated.

### 5.8.1 Example usage

```
>>> history = (('python', '⬛⬛⬛⬛⬛🟨'), ('debunk', '🟩🟨⬛⬛🟨⬛'))
>>> print_keyboard(history)

    Keyboard information
    ------------
    a:       b: ⬛
    c:       d: 🟩
    e: 🟨    f:
    g:       h: ⬛
    i:       j:
    k: ⬛    l:
    m:       n: 🟨
    o: ⬛    p: ⬛
    q:       r:
    s:       t: ⬛
    u: ⬛    v:
    w:       x:
    y: ⬛    z:

>>>
```

## 5.9 `print_stats(stats: tuple[int, ...]) -> None`

`stats` is a tuple containing seven elements, which are the number of rounds won in 1-6 guesses, and the number of rounds lost, respectively. This function prints the stats in a user-friendly way.

### 5.9.1 Example usage

```
>>> stats = (1, 2, 3, 4, 5, 6, 7)
>>> print_stats(stats)

    Games won in:
    1 moves: 1
    2 moves: 2
    3 moves: 3
    4 moves: 4
    5 moves: 5
    6 moves: 6
    Games lost: 7
>>>
```

## 5.10 `main` function

The main function should be called when the file is run, and coordinates the overall gameplay. The `main` function should be fairly short, and should utilize other functions you have written. In order to make the `main` function shorter, you should consider writing extra helper functions. In the provided `a1.py`, the function definition for `main` has already been provided, and the `if __name__ == __main__:` block will ensure that the code in the `main` function is run when your `a1.py` file is run. Do not call your `main` function (or any other functions) outside of this block. The output from your `main` function (including prompts) must exactly match the expected output. Running

the sample tests will give you a good idea of whether your prompts and other outputs are correct. See Figure 2 for example usage.

***NOTE:*** You **must** use the functions from `a1_support.py` to draw answers for each round from the possible answers. **Do not** import `random` or make any calls to anything from the random library directly in your own code; doing so will cause you to select the incorrect answer words in testing and receive zero marks for many tests.

# 6 CSSE7030 Task

CSSE7030 student must also add a solver to the Wordle game. At the prompt for a guess, if a user enter 'A' or 'a', your solver should assist them by making the next guess. The word chosen for the guess must not violate any known information from previous guesses (e.g. if it is known that a letter exists in a position, the guess must have that letter in that position).

NOTE: you ***must not*** use anything from the random library or the `choose_word` helper function from `a1_support.py` as part of this task, as doing so will cause your program to choose the wrong answer words for earlier tasks in testing. You may use any non-random method for choosing a guess from the valid candidate words. As long as your solver chooses a guess within a reasonable time that does not violate any known information from previous guesses.

Figure 3 shows the added functionality during gameplay.

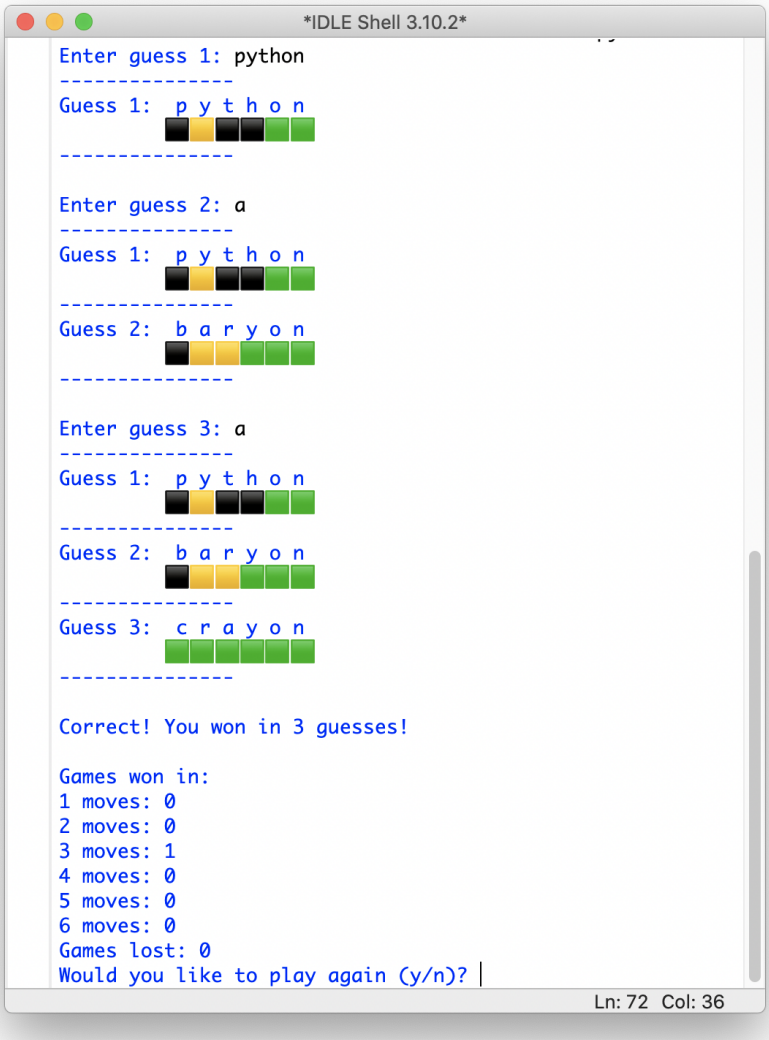



Figure 3: An example of the CSSE7030 functionality during gameplay. Your solver may give different answers, as long as they do not violate known information from previous guesses.

The design of the solver is up to you, but you *must* implement the following function, which should be called in your `main` function to handle the case where the user enters an 'A' or 'a':

## 6.1 `guess_next( vocab: tuple[str, ...], history: tuple[tuple[str, str], ...] ) -> Optional[str]`

Returns a valid next guess that doesn't violate known information from previous guesses. If no valid word remains in the vocabulary, this function should return None.

### 6.1.1 Example usage

```
>>> vocab = load_words("vocab.txt")
>>> history = (('python', '⬛⬛⬛⬛⬛🟨'), ('debunk', '🟩🟨⬛⬛🟨⬛'))
>>> guess_next(vocab, history)
'daines'
```

Note that while our implementation has guessed the word 'daines', other words, such as 'dancer' could be valid guesses in this situation. A word such as 'crayon' would be an invalid guess in this instance for each of the following reasons:

1. We know from the first guess that there is no 'y' or 'o' in the answer.
2. We know from the first guess that the 'n' in the word does not appear in the last position.
3. We know from the second guess that there must be an 'e' anywhere other than the second position.

# 7 Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. read and analyse code written by others,
3. read and analyse a design and be able to translate the design into a working program, and
4. apply techniques for testing and debugging.

## 7.1 Functionality

Your programs functionality will be marked out of a total of 6 marks. Your assignment will be put through a series of tests and your functionality mark will be proportional to the number of tests you pass. You will be given a *subset* of the functionality tests before the due date for the assignment.

You may receive partial marks within each section for partially working functions, or for implementing only a few functions.

You need to perform your *own* testing of your program to make sure that it meets *all* specifications given in the assignment. Only relying on the provided tests is likely to result in your program failing in some cases and you losing some functionality marks. Note: Functionality tests are automated, so string outputs need to match *exactly* what is expected.
Your program must run in the Python interpreter (the IDLE environment). Partial solutions will be marked but if there are errors in your code that cause the interpreter to fail to execute your program, you will get zero for functionality marks. If there is a part of your code that causes the interpreter to fail, comment out the code so that the remainder can run. Your program must run using the Python 3.10 interpreter. If it runs in another environment (e.g. Python 3.9 or PyCharm) but not in the Python 3.10 interpreter, you will get zero for the functionality mark.

## 7.2 Code Style

The style of your assignment will be assessed by a tutor. Style will be marked according to the style rubric provided with the assignment. The style mark will be out of 4.

The key consideration in marking your code style is whether the code is easy to understand. There are several aspects of code style that contribute to how easy it is to understand code. In this assignment, your code style will be assessed against the following criteria.

- Readability
  - Program Structure: Layout of code makes it easy to read and follow its logic. This includes using whitespace to highlight blocks of logic.
  - Descriptive Identifier Names: Variable, constant, and function names clearly describe what they represent in the programs logic. Do not use Hungarian Notation for identifiers. In short, this means do not include the identifiers type in its name, rather make the name meaningful (e.g. employee identifier).
  - Named Constants: Any non-trivial fixed value (literal constant) in the code is represented by a descriptive named constant (identifier).
- Algorithmic Logic
  - Single Instance of Logic: Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a function.
  - Variable Scope: Variables should be declared locally in the function in which they are needed. Global variables should not be used.
  - Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. loops and conditional statements).
- Documentation:
  - Comment Clarity: Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.
  - Informative Docstrings: Every function should have a docstring that summarises its purpose. This includes describing parameters and return values (including type information) so that others can understand how to use the function correctly.
  - Description of Logic: All significant blocks of code should have a comment to explain how the logic works. For a small function, this would usually be the docstring. For long or complex functions, there may be different blocks of code in the function. Each of these should have an in-line comment describing the logic.

## 7.3 Assignment Submission

You must submit your assignment electronically via Gradescope (`https://gradescope.com/`). You **must** use your UQ email address which is based on your student number (e.g. s4123456@student.uq.edu.au) as your Gradescope submission account.

When you login to Gradescope you may be presented with a list of courses. Select CSSE1001/CSSE7030. You will see a list of assignments. Choose **Assignment 1**. You will be prompted to choose a file to upload. The prompt may say that you can upload any files, including zip files. You **must** submit your assignment as a single Python file called `a1.py` (use this name all lower case), and *nothing* else. Your submission will be automatically run to determine the functionality mark. If you submit a file with a **different name**, the tests will **fail** and you will get **zero** for functionality. Do **not** submit **any** sort of archive file (e.g. zip, rar, 7z, etc.).

Upload an initial version of your assignment *at least* one week before the due date. Do this even if it is just the initial code provided with the assignment. If you are unable access Gradescope, contact the course helpdesk (csse1001@helpdesk.eait.uq.edu.au) *immediately*. Excuses, such as

you were not able to login or were unable to upload a file will not be accepted as reasons for granting an extension.

When you upload your assignment it will run a **subset** of the functionality autograder tests on your submission. It will show you the results of these tests. It is your responsibility to ensure that your uploaded assignment file runs and that it passes the tests you expect it to pass.

Late submissions of the assignment will **not** be marked. Do not wait until the last minute to submit your assignment, as the time to upload it may make it late. Multiple submissions are allowed, so ensure that you have submitted an almost complete version of the assignment *well* before the submission deadline of 16:00. Your latest, on time, submission will be marked. Ensure that you submit the correct version of your assignment.

In the event of exceptional personal or medical circumstances that prevent you from handing in the assignment on time, you may submit a request for an extension. See the course profile for details of how to apply for an extension.

Requests for extensions must be made **before** the submission deadline. The application and supporting documentation (e.g. medical certificate) must be submitted via my.UQ. You must retain the original documentation for a minimum period of six months to provide as verification, should you be requested to do so.

## 7.4 Plagiarism

This assignment must be your own individual work. By submitting the assignment, you are claiming it is entirely your own work. You **may** discuss general ideas about the solution approach with other students. Describing details of how you implement a function or sharing part of your code with another student is considered to be **collusion** and will be counted as plagiarism. You **may not** copy fragments of code that you find on the Internet to use in your assignment.

Please read the section in the course profile about plagiarism. You are encouraged to complete *both* parts A and B of the academic integrity modules *before* starting this assignment. Submitted assignments will be electronically checked for potential cases of plagiarism.

## 7.5 Changelog

**14 March 2022** Changed due date from 25 March 2022 to 1 April 2022 and amended typo stating that assignment should run in Python 3.9.