Scrabble Recreation

The purpose of my project is to recreate the popular board game Scrabble. Allowing between one and four players to play against each other or against themselves, turn by turn, using the seven letters they are distributed to create the highest scoring word possible.

Every word placement, excluding the first word placed (in the centre of the board) must connect with another, meaning players must make use of surrounding letters to form their word, otherwise the word cannot be placed. After each successful placement of a player's word, the scoring of each tile is then summed, and any bonus spaces on the game board are taken into account, this total is then appended to the players score.

Replacement tiles are then distributed back to the player, for the sum of seven. However, the amount of letter tiles available to be distributed becomes gradually lower as the game progresses, and so once a player has used up all their tiles and there are no more available to redistribute, the game will end and the player with the greatest total score will win.

Contents

What makes Scrabble programmable? 4
What language will be used? 4
Who will use the program? 4
Research of active recreations and notable features 5
Initial success criteria 8
Initial project scope 8
Development cycle 9
Further notes and ideas 9
Method of play 10
Stakeholder feedback 11
Finalised success criteria 12
Hardware and software requirements/notes 12
Decomposition 13
Design 14
Analysis of individual components 15
Gantt Chart 16
Variables and data types 17
Simplified pseudocode of the whole game logic 18
Designing game systems 19
Method dependency / Validation considerations 19
Test data 19
Post development 19
Development 20
Sub system – Turn based input 20
  Development – Turn based input 20
Sub system – Distributing and removing tiles 22
  Development – Initial distribution 22
  Development – The Dealer 23
  Development – Removing tiles 24
Sub system – Array to grid display 26
  Development – Array to grid display 26
Sub system – Grid based word placement interface 28
  Development – Coordinate validation 28
  Development – Converting the coordinate to an index 29
  Development – Placing the words 31
Prototype – Turn based word placement 34
Full Prototype – Turn based word placement 40
Sub system – Placement validation 47

Development – Copying the board 47
Development – Validation 49
Development – Word validation 50
Development – Before word validation 51
Sub system – Grid slicing 51
Development – Grid slicing 51
Development – Initial word input validation 54
Sub system – Managing user inputs 55
Development – Managing user inputs using classes 56
Development – Sandbox test environment 60
Resumed Development – Before word validation 61
Sub system – Within word validation 63
Development – Within word validation 63
Sub system – After word validation 64
Development – After word validation 64
Sub system – Dictionary check 66
Development – Implementing a dictionary 66
Development – Comparing a word to the dictionary 67
Sub system – Owned tiles 67
Development – Owned tiles 67
Sub system - Surrounding word validation 69
Development - Surrounding word validation 70
Issues to resolve 72
Overwriting tiles 72
Filtering words formed 73
Dictionary checking 74
Sub system - Scrabble rules 76
Development – Passing through the centre 76
Resuming development – Passing through the centre 77
Development - Connected words 78
Sub system – Bonus tiles 79
Development – Bonus tiles 80
Development – Detecting bonus tiles 81
Sub system - Scoring 82
Development – Tile scores 83
Development – Word scoring 83
Development – Calculating the score 86
Reference base tile scores (and distributions) 88
Limitations in scoring and other issues 89
Stakeholder feedback 89
Development - Tracking scores 90
Sub system – Win condition 90
Development – Running out of tiles 90
Development – Forfeiting the game 91
Development – Final scoring 92
Development – Determining the winner 93
Sub system - Final improvements 93
Empty inputs 93
Output display 95
User guide 96
Finished product 98
Full code 98
Gameplay output 108
Evaluation 115
Bibliography 123

This page is intentionally blank.

Luc Heaton

What makes Scrabble programmable?

There is a set of defined rules that do not change as the game progresses, for example, every word placement is scored in the same way. However, the resultant score is dependent on the tiles used, the location at which it is placed, the score of any other words that may be a result of the combination and if all the tiles of a player are used in one turn, scoring a bonus fifty points.

Due to the fact the game revolves around grid-based play, a two-dimensional array will be a perfect representation of the playing board, as they can be easily traversed through code, allowing the manipulating of values as the game progresses, such as updating the position of letters within the array. After its initialisation the array size is fixed, meaning elements are not deallocated even though they may not have a value.

The turn-based gameplay can be represented well computationally, as for each turn the player has the same set of decisions and inputs. For example, every turn a player must either create a word or pass. If a word is created it must go through a set of defined verification and validation methods, this could be processed as; each input being parsed into a selection of functions, which may include, checking the word against a dictionary, then checking the placement, finally, determining if the word is valid or not. This process is the same every turn, which means it can be programmed as a loop.

The game is heavily dependent on word creation and combination; this can also be programmed through string manipulation and concatenation. However, knowing this process is required influences which programming language I will use to create my project, as some languages handle string manipulation much better than others, such as Python, compared to C#. Python has loads of in-built methods when using strings, implementing the same functions within C# will require greater time and may be harder to manage.

What language will be used?

I have chosen Python as the most suitable language to program Scrabble, as the large majority of the project will be string and array manipulation, because of the main functionality of placing words, making use of other letters within the game board, grabbing surrounding letters and checking validity of the placements. It is because of this repeating process that it can be achieved computationally.

Improving my knowledge surrounding Pythons string and array manipulation functionality will be very useful in the completion of this project. Learning how to effectively make use of the large selection of available functionality that Pythons libraries provide.

Who will use the program?

The users of my program will most likely be those looking for a version of Scrabble; that is playable on most devices (given they have a Python interpreter) that is not dependent on any external libraries, assets or an internet connection, connecting the program to a server.

That being said, it is possible to compile Python code using tools such as Thomas Heller's py2exe, which unfortunately is dependent on Windows architecture, making it platform specific. However, there is Gordon McMillan's Installer, which allows cross-platform execution allowing any device to execute the program in a command line interface.

The main users of this piece of software will mostly be casual players, that have a good understanding of how the game of Scrabble is played, meaning they do not require an explanation of how to play the game. Because of this assumption, I will not be focusing my efforts on producing a method of teaching players how to play the game, as with a basic understanding and a clean interface, it should be self-explanatory to utilise. However, I may have to include small tool tips, showing the formatting of input.

Research of active recreations and notable features

When analysing previous recreations or attempts at programming Scrabble, I have noticed many developers create programs that only allows players to play against themselves, where I plan to develop a multi-user game allowing multiple players to play against each other, using the same device. Yet, also allowing players to challenge themselves in improving their skill in word formation.

I have also seen a few multiplayer versions of the game allowing networked gameplay; however, I am not intending to produce this kind of experience. It would take a significantly greater amount of time to produce, and a lot of work would be required just to send validated data to a server where it would be verified, plus taking the chance of data loss during transmission into account, let alone managing the game state. Which is why I plan to create a local multi-user interface.

The game requires a set of letter score values, which are used to determine the score of a placement. When reading other projects, some define their own set of tile scores. However, using my own for the tiles is not worth working on with my limited time, so instead, the scores assigned for each tile will most likely be the scores used by the official game, as they are well defined and have been adjusted to create the most balanced gameplay.

The main playing board or grid would most likely be a two-dimensional array, because each element could be interpreted as a cell by the program, and the number of cells is fixed e.g. the game grid in Scrabble does not change as the game progresses. In contrast, I could use multiple arrays for each row/column of the grid, which may make it easier to understand the code; however, the same result could be achieved with fewer variables.

The main limitation to this project is the achievable graphical interface in the time allocated. My main options for representing the game to the user are a text-based grid compromised of ASCII characters, which may be easier and take less time to refine and finish. However, a graphics-based system could be produced instead, making use of images and buttons, which would be more user-friendly, but require a greater amount of time to create.

Luc Heaton

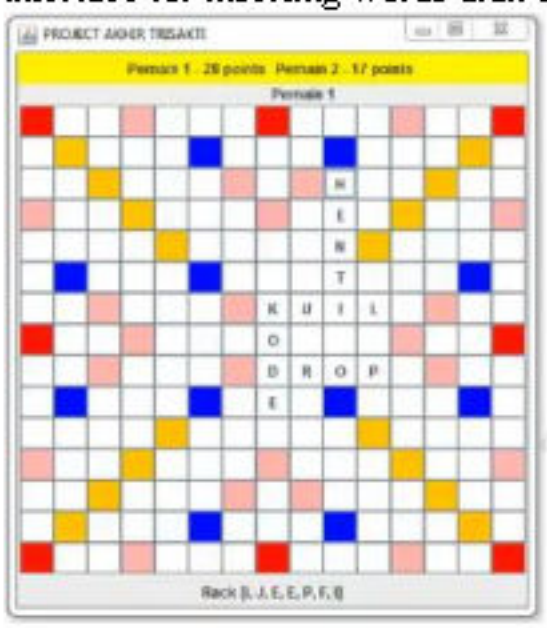
One solution I found to Scrabble was developed using C# and Windows Forms. Making use of WinForms for a simple GUI. From the description and some media showing the games functionality, such as this one. It resembles the word placement and simplified tile scoring, and some other features too.

However, it does not really incorporate the rules of Scrabble, such as having seven available tiles and turn based player gameplay.

After, looking into it. The project is more of a proof-of-concept for placing letters with an assigned value onto a grid and scoring them. Which works well. The main take away of this project is the usage of colours to represent letter scores, making it clear to user which letters are more valuable.

As much as this is a clear, simple word game. It's not really Scrabble, and with my project, I plan to produce a solution that resembles the game as closely as possible in the allocated time.

I found a few projects developed in Java. The first one made use of a simple GUI like the former, but the interface for inserting words didn't seem very convenient, opening up an external window to give a user-input.



Even though, the game is developed for a different language, the features of the game stay the same. In this project, the user selects a starting tile then by using the external window, the player can enter a word and enable a value to determine the orientation of the word.

The final version shown includes just placing of words. With no scoring in place and it seems that there is no validation for the words being placed and you can place the words without having them within your "Rack". The UI elements just look like static placeholders that are yet to be developed.

Scrabble game in C# by LiorBobby

As well as finding solutions to the whole game, I also found many tools acting as Scrabble Helpers. That given some rules and the tiles available to a player, the program will output the highest scoring words that could be played. Reviewing these may be useful, as, to successfully place a word within the game I must essentially complete the same validation methods as these tools do, to review if a placement is valid.

A large portion of these tools are developed using JavaScript as they are embedded within websites, making the convenient for a player to use at any point. Most of them work based on wildcards, allowing a user to define the letters at the start/end of a word, requesting an output that fits the given criteria.

www.wordplays.com

This is an example of one embedded in a website. Which even has an "Advanced" option allowing users to enter more parameters to narrow down the search.

Scrabble by Abinaya Rajesh

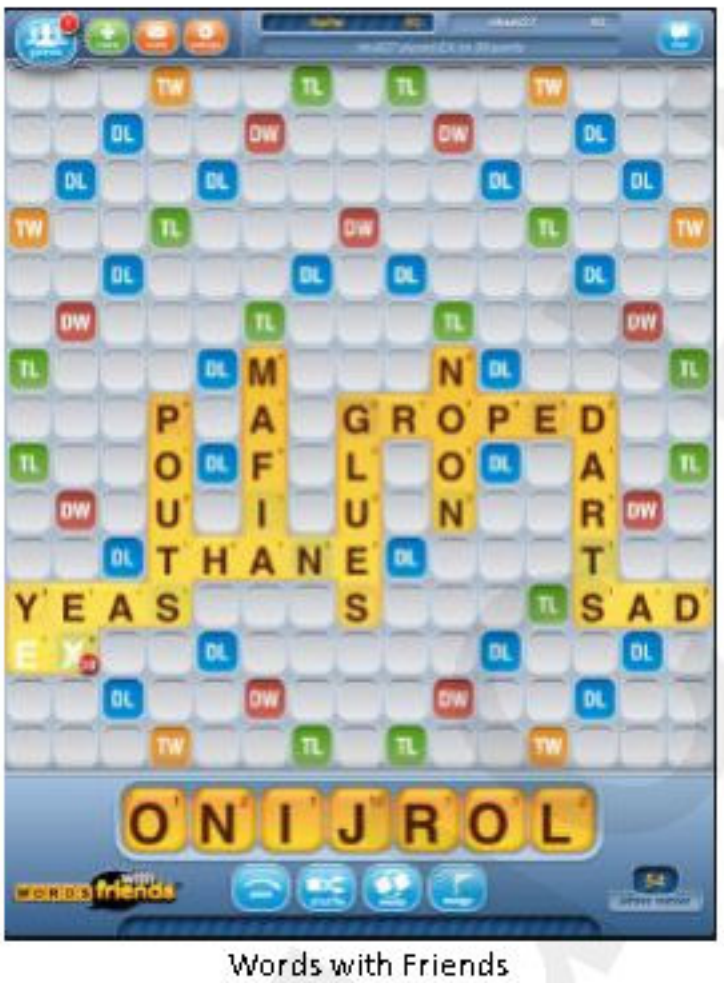
One of the best solutions I have found so far, for something I may be able to achieve within the allocated time, is this project developed using Python and Tkinter, which is a package for Python allowing the implementation of GUI elements.

The program allows users to drag and drop their given tiles onto a game board, making use of bonus tiles and surrounding tiles.

Once, the player has finished placing their tiles, they can click the "Make Move" button, which will validate the placement, score the player and then end the turn. A huge feature of this project is that they have programmed a method of the computer creating and placing words, so the player has something to play against. In addition, there seem to be other features such as Shuffle, Exchange and Hint, which were not demonstrated, however I assume they do their expected function.

Either way, this solution is useful as it displays the game in a working state, proving it is computationally possible with the tools available to me. Sadly, the programmer did not provide the source code or any information regarding the project, as I would really like to know what their solution is to a computer player.

The only element missing from this project is allowing multiple user-controlled players to interface with the game as well, however, that may not have been part of their design brief.



Words with Friends
by Zynga

There are also many other products, that provide a better Scrabble experience in the form of apps and web-games that are developed by larger teams. Which include online player vs player games, player vs computer, challenges and even player rankings allowing users to exceed their friend's high-scores.

As much as, producing a project of this standard is feasible. I do not have the knowledge to produce a networked game of this scale. However, it includes the features I need to work towards.

Initial success criteria

To accurately recreate the game of Scrabble, and for my project to be successful, my program may include or allow:

- Tiles with an assigned letter and score, allocated to the player, allowing them to create words.
- A virtual playing board with a limited number of spaces for tiles, with bonus score values distributed around it, creating opportunities for the play the score extra points.
- A virtual bag containing a finite number of tiles that is interfaced with by the dealer throughout the game.
- A dealer that "randomly" distributes tiles to players at the start of the game and at the end of each word placement, meaning each player has seven tiles to utilise each turn. Up until the bag becomes empty and no more tiles can be distributed as replacements after a placement.
- An interface that allows users to input a word based on the characters available to them (including blanks), defining the position and orientation of placement.
- Instructions or helping tips to show a user how to use the software.
- Placement verification methods
    - Preventing the overwriting of tiles during placement.
    - A system to check word(s) validity against a dictionary.
    - Prevent word placement outside of the grid without causing errors.
    - A method of grabbing surrounding words, if a word placement forms more than one word, it must check if they are also valid words.
- A method that detects if a player has used all their tiles and there are none left to distribute.
- A leaderboard that shows players and their score at the end of the game.

Dependent on the time allocated to the project, less fundamental features may not be included as it is may not be a feature that is required for the game of Scrabble to work, for example a hint system that will assist the player in some manner. On the other hand, some features may be added or improved, for instance utilising some graphics, to make the interface more appealing, which would be a quality-of-life improvement.

Initial project scope

To further illustrate the elements that must be developed in-order for the project to become a working solution. I have separated the features that are within scope, and must be developed, from the features that are out of scope, meaning they are either, not required for the final solution to work, or, they are quality-of-life elements, which have a lower priority for development.

In-Scope	Out-of-Scope
- A hardcoded list of tiles with an assigned letter and value. Including blanks. - A grid that displays the current state of the game. That displays coordinates, and bonus scores. - A virtual bag containing the hardcoded list of finite tiles, it is deducted from as titles are used. - Random distribution of tiles from the virtual bag, at the start of the game, and as tiles are placed. - A text-based interface-allowing users to enter the data required to make a placement. - Validation preventing players from entering data that will cause the program to break. - Validation of each placement, including but not limited to: valid English words and placement within the grid range. - Correct scoring of placements, similar to the official game. - Detection for the end of the game, ending the game and displaying the highest scoring player.	- A set of instructions or helping tips that will teach a player how to use the software. - A graphical user interface to replace the text-based interface, making it clearer and easier to interact with the program. - A leaderboard displaying each player, in order of score. - A hint system that could form words using the player's tiles, at some cost. - A shuffle action that shuffles a player's tiles that may aid them to form words. - Allowing players to exchange an amount of their tiles, swapping them for random tiles within the virtual bag (if available), at the cost of their turn.

Evaluating, the projects scope, I have placed instructions in out-of-scope because, despite the high requirement for them (so that users understand how to use the program), it is of higher priority that the program is developed and working first. Then quality-of-life features that are out-of-scope may be implemented. The same can be said for the other features that are currently out-of-scope. This is mainly dependent on time limitations for the completion of this project.

Development cycle

The model of development that will be suitable for the production of this project is Agile, due to the limited amount of time allocated and the requirement for a product that meets as many specification points as possible, which is achieved by the continuous testing of systems for the project through its development. Agile development will work well with Python because it is an interpreted language, and so I will not be required to compile my project each time I would like to test newly implemented code, which may slow down development time.

Further notes and ideas

Each unique tile must be declared within the code. Each character within the English alphabet will have a tile assigned to it, alongside an integer score associated with it. In addition, blank tiles will be included, meaning the tile can be assigned any letter of the players choosing – these tiles have no score assigned to them. When a group of tiles are placed and form a valid word, the sum of the tile scores is appended to the placers score, plus any bonuses.

As mentioned before the playing board could take the form of two-dimensional array, as each space of the board can be represented within the array. From the user's viewpoint, the array will be shown as a grid of defined size of equal side length, e.g. 15x15. The grid will have to include bonus spaces, where extra score is awarded when tiles are placed on them, for example a space on the grid labelled as TLS (Triple Letter Score), awarding a tile of base score three placed on it, a score of nine.

Spaces on the playing board will include bonus spaces such as TWS, TLS, DLS, and DWS. L meaning Letter, W meaning Word, T and D meaning Triple and Double respectively, rewarding players for creating words over them. Also, the centre marking space at the centre of the board, where the first word of the game must be placed.

The virtual bag must contain a set number of tiles, which are distributed to players throughout the game, allowing the playing board to fill up by a sufficient amount, increasing the difficulty over time.

The "dealer" may be a function that, dependent on the number of tiles missing from a player's default set of seven will distribute the tiles required for a sum of seven. For example, given a player only has a set of five tiles, he is missing two, the dealer function is required to deal to randomly selected tiles to the player, removing them from the virtual finite sized bag.

For players playing alone, or wanting an extra opponent I could implement computer-controlled players, which will have two methods of creating valid words. The first could include the analysis of its dealt tiles by comparing every combination of tiles against a dictionary file, creating a list of valid words. Then searching through the playing board for a space so that its placement does not collide with other words.

The second method could be a variant of a brute-force search, in which a function would look at every cell in the grid and attempt to place a word in both orientations; it would do this for every cell until it found a valid placement. Some improvements to this idea may include storing multiple valid positions and selecting the highest scoring one, allowing the function to break after a set number of passes, to prevent the function being stuck in an infinite loop, when there is no possible word to be formed.

Throughout this project, I may interchange letters and tiles, as in the context of Scrabble they are regarded as the same. Tiles being the physical pieces that are placed within the grid that represent a letter with an associated score or value.

Furthermore, I may interchange the data type "array" and "list", which technically cannot be done as they are fundamentally different, arrays are of fixed size and cannot change in size after initialisation, whereas lists are dynamic and their size can be changed at runtime. I am making this statement as I am using Python, which does not include arrays, only lists. So, for the example of the game board which is always a fixed sized

variable I may reference it as being an array as its size will not change, whereas within the program it would take the form of a list.

Method of play

To gain a better understanding of the interface that I will be producing, allowing a player to interact with the program, I am going to consider the method(s) in which I think users could use the program and some of the features of Scrabble I must take into account. For example, the method in which a user can specify and make a move on their turn, such as the location of placement.

The first interaction that a user must make is the input of the selected number of players that will be taking part in the game. This should simply be an integer, with a minimum of one and a maximum of four. Technically, I could make the maximum larger; however, too large a number of players would mean that it is possible to win the game before it has even begun. As, if there were more players than tiles to distribute, for seven tiles each, a player (or multiple), would meet the standard Scrabble win condition (of using all your tiles with none left to distribute from the bag). On the other side of the spectrum, the minimum value should allow a single player to play against themselves. In summary, I think sticking to the standard number of players for a board game will be suitable for this solution.

The second important interaction would be the input of player names, which will be referenced throughout the program, and at each turn, making it clear whose turn it is. I feel as though, users should have complete control on the alias they want to have assigned to them, whether it's lowercase/uppercase, contains symbols or numbers. As it would not affect the execution of the program in any meaningful way. My only further thought and slight concern that I have with this method is that having no limit to name length, would mean it would be possible to have a wall of text as an alias. Although, it would not cause any issues in the program's execution, it may cause a slight inconvenience to the players.

After assigning each player an alias, the next input that would be required by the user would be for the first players turn; entering the word that they would like to form using the tiles that were distributed to them at the start of the game. The first consideration is that the capitalisation of this input should have no impact on the program. For example, the strings "HeLLo" and "hello" should be equivalent, giving the same result. Continuing with this, as it is the first turn of the game, the word does not need to combine with another word, as there are no tiles on the grid to combine with, so the check for a combination, which would be made for each subsequent placement, would be ignored on the first turn.

Furthermore, as this is the first word of the game the word must be placed at the centre position of the game grid, meaning that this player would have to enter a position and orientation that passes through the centre, otherwise the input should be classed as invalid.

At any point, if any input is invalid, such as a word that cannot be constructed using a player's tiles (and the available tiles on the gird), or if the inputted placement position is invalid, as examples. The turn should be reset, allowing the player to try again, re-entering their inputs. However, there should also be the option if the player cannot make a move in which they can pass, forfeiting their turn.

In the example that a player possesses a blank tile, I want to produce a method that simply receives the input of a word (containing the tile they do not own) in the same way as every other turn. Yet, the program detects that the player has formed a word using a letter that is not available to them, also detecting that the player has a blank tile available, so the method will simply remove the blank tile from their inventory. However, if a player uses a tile they do not possess, without a blank available. The program should state the issue and allow the player to try again.

Lastly, in all further turns. Each word must connect to another, making use of the available tiles on the grid to form a word; otherwise, the placement is not valid. To achieve this, I have to decide how a player formats their word input, declaring that a tile from the board will be used, to form the complete word. Considering my solution above for the usage of blanks, I think that the simplest solution would be for the player to enter a space character in place of the tile used from the grid, for example, to place the word "FAR", combining it with an "A" character that already exists on the board. The user would enter "F R", indicating that the second character is a placeholder for a tile that is already on the board.

Stakeholder feedback

Addressing the considerations in order, above. For the range of players that will be playing the game at a given time, I agree that it would be suitable to stick with similar ranges used for common board games. As it would play more similarly to the original board game, which is the aim. Meaning that the game will last for a standard amount of time, relative to a board game, giving each player a full experience in forming and placing words. Furthermore, I like the idea of allowing a single player to play against themselves, as logically there should be no issue with that, i.e. the game would play in exactly the same way, except each round it would be the same player's turn, and the winner would always be the same person. But, then again, a person playing alone would most likely be playing to improve their word construction and score.

Moving to the next point, I also agree with the freedom of player name choice. As this game will be played sharing the same device, meaning that if there are multiple players they will be sharing the same computer, swapping out each turn, for example. This means that the players will most probably know each other to some extent, so the issue of offensive names is up to the discretion of the players involved, for instance if an older player is using the program alongside a younger person, it would be their choice in whether to use a questionably inappropriate name, when with a younger impressionable person. However, if this was a multiplayer/online game, I would want some filtering to the aliases of players, as each party would not necessarily know each other.

However, in the case of name length I suggest there is a character limit. As I think that it is unreasonable to have a twenty character or greater length username, I would not like to see a large "wall of text" as a player's username.

I agree that the words entered by a user should be independent of the capitalisation of the letters entered, it would make the user experience simpler, tolerating for the slight error of capitalisation. Otherwise, it would become another inconvenience to players.

I am in favour of the idea of users being allowed to re-take their turn, if any input they enter is invalid, as it would further enhance the user experience, in that users don't have to be as careful with their inputs, to achieve the output they intend to have. As well as an option to pass their turn - I think that this is a required feature, as it is possible and quite common within the original game to be left with a selection of tiles which you cannot form a word with, and so the only option is to (pass and) wait for another player to create an opportunity for you to create a word with. In addition to this, I think that a condition that detects if all players have passed their turns (meaning that nobody can make a move) consecutively, should be incorporated as a second win/end game condition, otherwise there could be a scenario in which nobody can make a placement, yet the game won't end.

Again, with the usage of blank tiles, it is a discussion of user experience. Your method of detecting the usage of blank tiles seems fairly intuitive, as it wouldn't require the user to perform any extra actions on their turn, which I am in favour of.

Lastly, regarding your proposed solution for combining placements by using a space in the position(s) of the tiles being substituted for ones on the board. I feel as-though it would make more sense for a user to enter the tile they are planning to use from the grid, in a similar method to the usage of blanks. On the other hand, it is still understandable for a user, and in some aspects, it may be easier to interpret the intended action that the player is making, for both the player and the program.

Finalised success criteria

After receiving feedback based on the initial success criteria, I have decided to re-evaluate the criteria that should be met, in-order to deliver an accurate recreation of Scrabble.

- The game is played by up to four players.
- The gameplay should not change depending on the number of players.
- The game will be turn-based.
- Each player should be allowed to assign themselves an alias.
- There should be a character limit for the aliases.
- The interface for user inputs must be validated.
- The capitalisation of user inputs should have no effect.
- A player should be able to reattempt their inputs, before their placement has been made.
- A player should be able to forfeit their turn, allowing the next player to play.
- Where each player has forfeited their turn consecutively, the game should end.
- There will be a finite number of tiles available to use throughout the game.
- Each tile has an associate tile score.
- Blank tiles that have no score value, but can imitate any alphabetic character.
- There should be bonus tiles distributed around the game board.
- The display will show the current state of the game, including placed words and bonus tiles.
- At the end of a player's placement, they will be correctly scored, depending on the tiles used, any surrounding tiles, and any bonus tiles, similar to the original game.
- Instructions to help the player understand the method of play.
- At the end of the game the winner is the player with the highest score.
- The game should follow the original Scrabble rules.

Hardware and software requirements/notes

As for the considerations regarding the medium that the product will be used on. The finished product will have been developed in Python, and because of this, the program is not limited to my current platform for development, being Windows. However, this also means that the program will be developed as a script, meaning that, for a player to use the program, they must have a recent version of the Python IDLE installed (as of writing 3.5.2). This clearly reduces the audience that this project can reach, due to this requirement. Although, a post-development could include the compilation of the script, creating a build for each platform, such as Linux, IOS and Windows, whereby only a single executable file is required to use the program (and maybe some libraries).

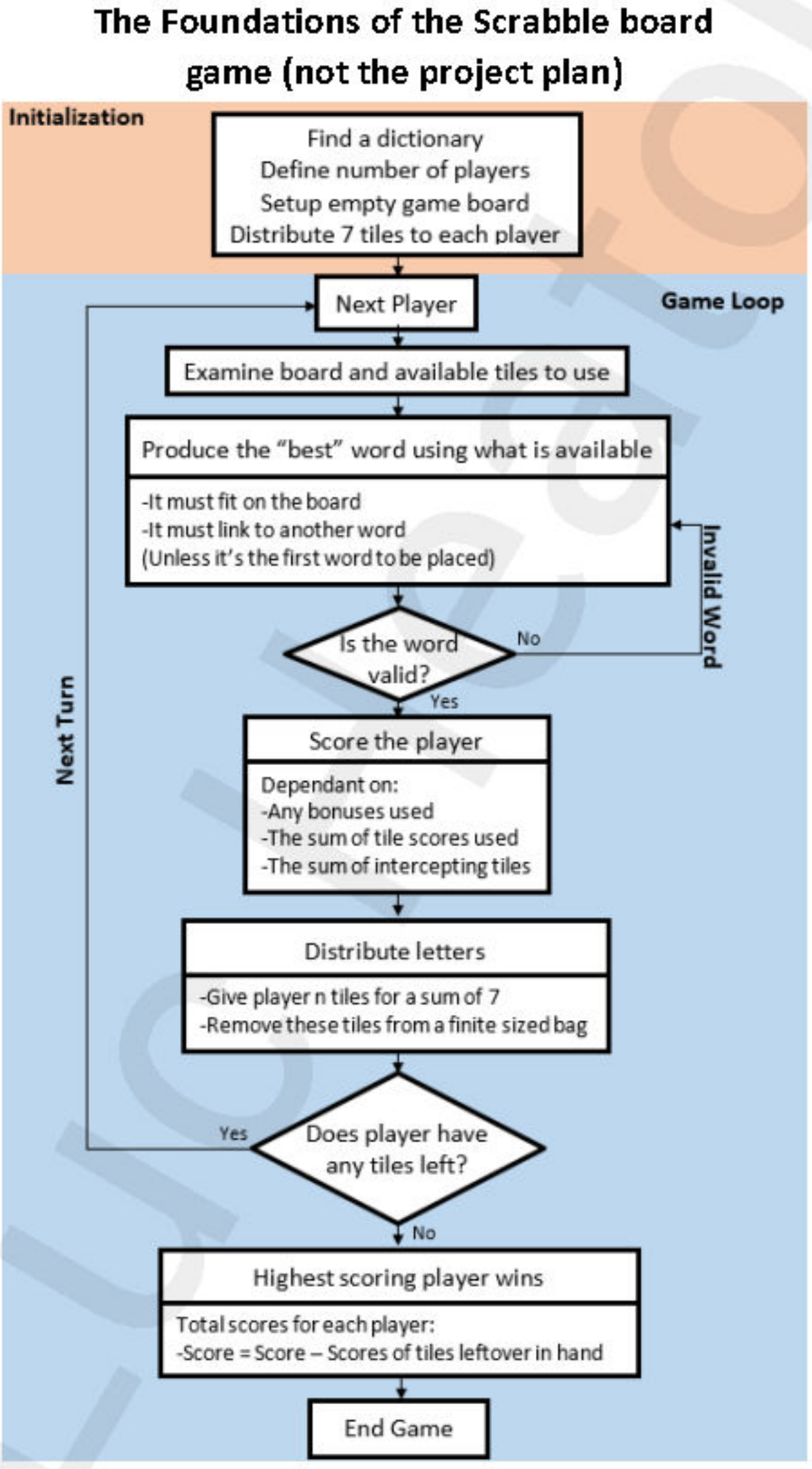
In the case of hardware. As mentioned, I will be developing this program as a script for the Python IDLE, meaning that in-order to utilise this program, the user's device must be capable of executing large Python scripts. Requiring roughly: a minimum of 2-4Gb, accounting for modern operating systems and background tasks. However, the discussion of hardware is fairly insignificant, considering the modern standard of processing power, also, the fact that this is a relatively small project, not requiring float point arithmetic or heavy graphic processing.

Luc Heaton

Decomposition

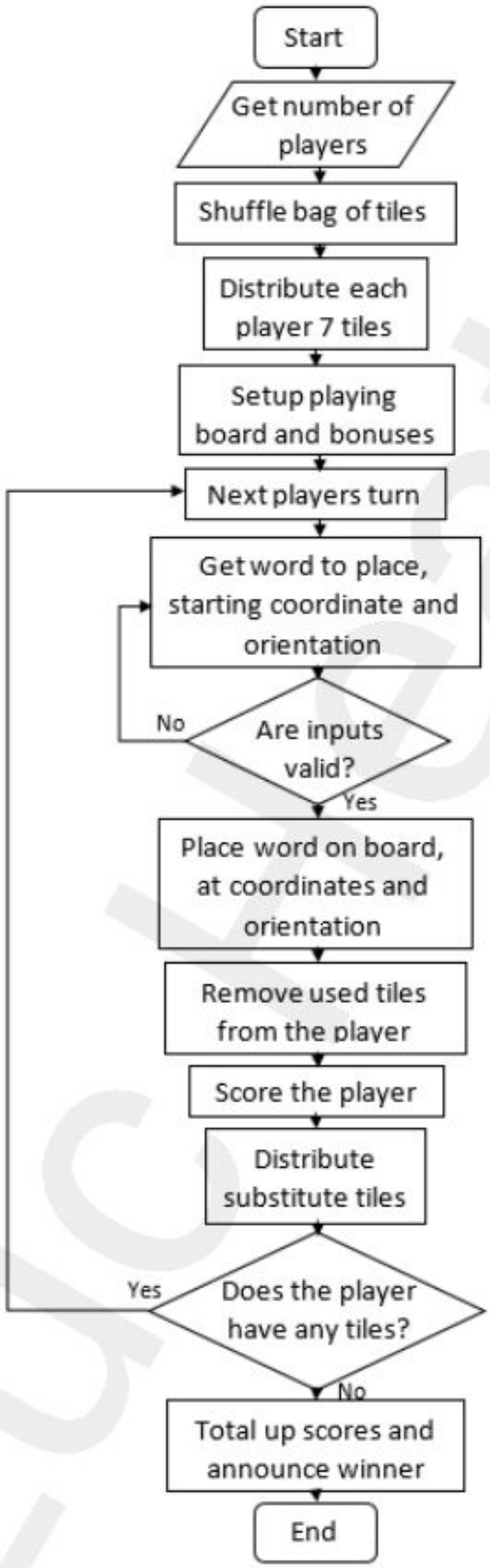
To start with, I decompiled the original board game version of Scrabble into its foundational game loop as well as some initialization that is required for the board game and similarly for my project.

However, this flowchart I produced illustrates the foundation that I must work towards (a simplified version of scrabble). For each process within the graph when developing the main program, I may be producing smaller sub-processes as it would be daft to fit each process within one function (in some cases).



Design

To begin with, I have produced a flowchart, which breaks down the program in a very general way. Overlooking the majority of the processing behind the placement of a word, for example. However, from this diagram I can start to design the computational logic for each of the condensed systems within it.



Luc Heaton

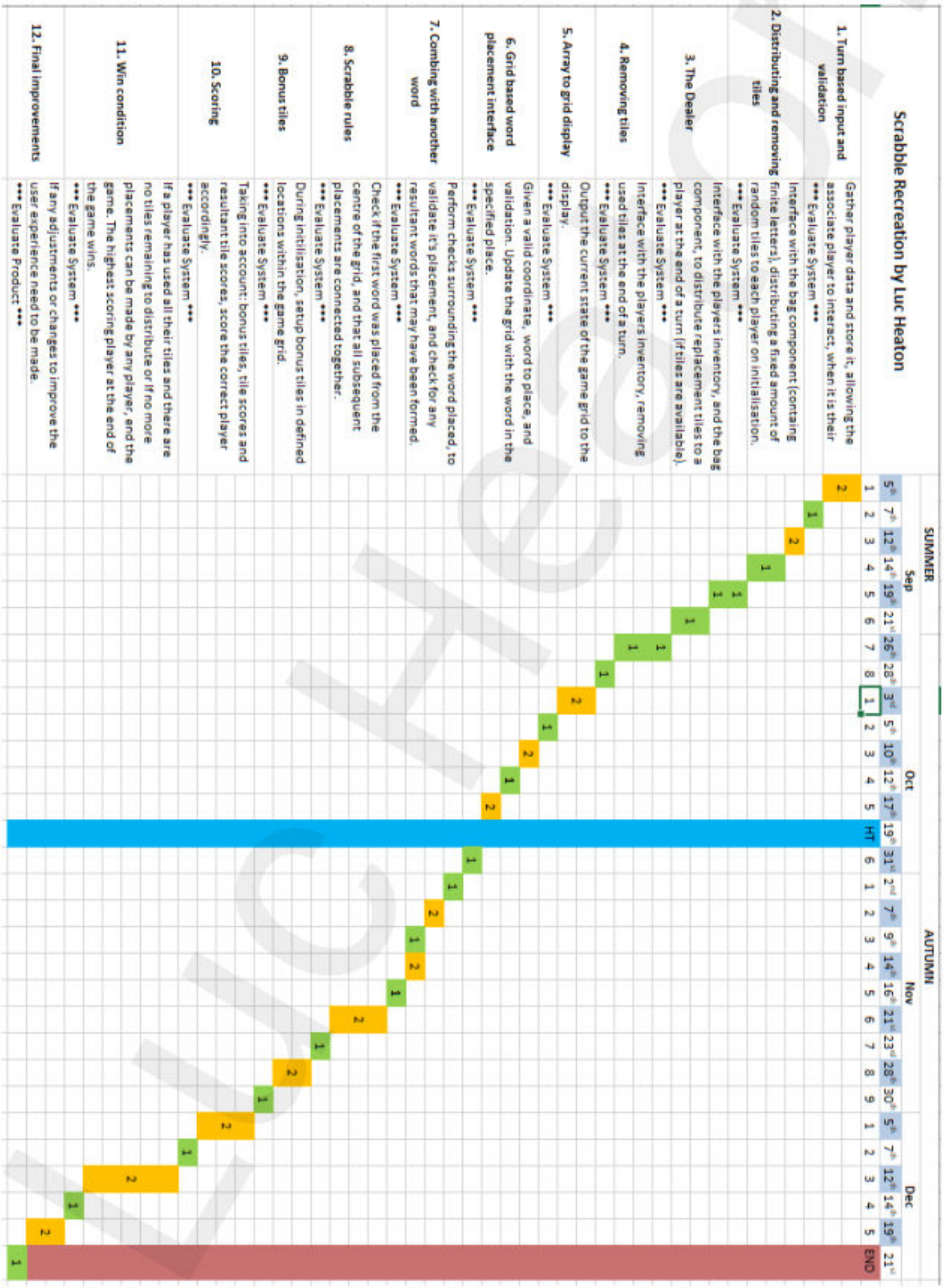
Analysis of individual components

Taking the individual systems that make up the diagram above, I composed a more detailed breakdown of each function, picking up on some or most of the functionality that each system must include.

1.Get number of players	2.Shuffle the bag of tiles
-Allow input of an integer in the range of 1 to 4 players. -Allow the user to give a name to each player. -Assign each user 7 tiles.	-Using a defined set of finite tiles, shuffle them, so that the distributed tiles are different each turn, each game. -Allow the collection of shuffled tiles to be removed from, so that eventually there will be no tiles available to be distributed to players.
3.Setup playing board and bonuses	**4.While no-one has won**
-Create empty playing board with bonus spaces such as TLS and DWS. -Display the board and update it when a new word is created. -Display coordinates along both axes.	-If the win condition has not been met, continue the game loop.
5.Next player	**6.Get input word, coordinates and orientation**
-Cycle to the next player, showing their individual tiles and score. -After reaching the last player in a round, loop back to the first player and continue.	-Request input of a string for the word input. -A letter followed by a single or double-digit number for the coordinate. -A Boolean to represent the two possible rotations: right or downward.
7.Are inputs valid?	**8.Place word on board**
-Validate if the string is creatable from the player's letters, but if a blank is available substitute it. -Verify if the coordinates are in range of the grid board.	-Validate the word does not overwrite any others. -Validate if the word is an English word, and any others that may be created in a single placement. -Validate the word is connected to another, except the first placement.
9.Score the player	**10.Remove used tiles from the player**
-Calculate the sum of tiles used in word placement, and apply any bonuses that may have been used. -Add calculated score to individual players score.	-Remove the tiles used to form the successfully placed word from the player, so that those tiles are out of play.
11.Distribute new tiles	**12.Does the player have any tiles?**
-After a player has placed a word, distribute tiles from the bag to a sum of seven. -If there are no tiles in the bag, do not distribute any.	-If after being distributed replacement tiles, the player still has no tiles after their placement, they have finished the game.
13.Total up scores	**14.End**
-The player with the highest score at the end of the game is announced as the winner.	-The program can either be reset to allow another game to be played, or the program will be exit.

Knowing that I must produce and document each of these sub systems, I have produced a Gantt chart, where I have made estimates as to the amount of time each system will take to develop, allowing myself an hour or

more at the end of each system to evaluate it. This chart only considers the working days I have to work on this project, when in reality, systems and features may take more or less time than expected. This is why I have allocated an amount of leeway to the production of each system, accounting for features that take longer than expected to develop. However, in the best-case scenario, if most elements of the project are developed earlier than expected, and I spend time outside of work hours, I may be able to finish the solution to the project in mid-November.



Gantt Chart

Variables and data types

With the diagram and analysis of the larger components I need to design and develop, I can define some of the essential variables that may be used throughout the execution of my program. Including data types, whether or not they are constant or dynamic and what they may be used for.

Even though I intend to program my solution with Python, the data types will be more specific. For example, there are no arrays built-in to Python, only lists, which do not have a fixed size. The same can be said for chars, only strings. The "Is Constant?" field is an indicator of if the value changes after initial assignment.

Variable Name	Data Type	Is Constant?	Notes
numPlayers	Integer	/	Created: By the user at the start of the game. Used: To get player names, and to track when the next round begins
playerNames	String Array	/	Created: By the user at the start of the game. Used: To store and track the active player's names. Mostly, used for display purposes.
playerScores	Integer Array	X	Created: Dependent on numPlayers. Used: Updated to store each player's current score will be used to determine winner.
playerInventories	2D Char List	X	Created: Dependent on numPlayers. Used: Stores the tiles available to each player at any given time, for example the first element will contain player one's tiles. It is a list because at a certain point the player will run out of tiles.
numTilesPerInv	Integer	/	Created: Hardcoded value. Used: As a constant o store the maximum number of tiles a player can have at any time.
availableTiles	String List	X	Created: Hardcoded values. Used: To act as the bag, storing all the tiles that will be distributed throughout the game. It is a list because as the game progresses the number of elements will decrease.
gameGrid	2D Array	X	Created: On initialisation, setting up default blank spaces and bonuses. Usage: Stores placed words and it is displayed to the user as a grid. Each element within the array stores a row of the grid.
currentTurn	Integer	X	Created: Before the game loop starts. Used: Accessing each player's data and being incremented at the end of each turn.
nobodyHasWon	Boolean	X	Created: On initialisation as True. Used: Is the condition that controls whether or not the game loop will execute again, only modified when the win condition is met.

Simplified pseudocode of the whole game logic

```
numTilesPerInv ← 7      //Define the number of tiles available per turn
playerInventories ← []  //Stores all active player tiles
playerScores ← [] //Stores all player scores
availableTiles ← ["A","A","B"...]   //Defines all the letters available
playerNames ← []      //Stores all player names
availableTiles ← Shuffle(availableTiles)  //Randomly shuffles all tiles
numPlayers ← USERINPUT "Enter number of players"
FOR x in range numPlayers DO  //For each player
      //Ask for a player name
      playerNames[x] ← USERINPUT "Enter player "+ x + "name"
      tempPlayerInv ← []
      //For the number of tiles available per turn
      FOR y in range numTilesPerInv DO
            //Pick a random index position of a letter
            randomIndex ← RandomInt(0,availableTiles.Length)
            //Add it to a temporary variable
            tempPlayerInv.Add(availableTiles[randomIndex])
            //Remove it from the list of available tiles
            remove availableTiles[randomIndex]
      //Assign the player the picked tiles
      playerInventories[x] ← tempPlayerInv
      ENDFOR
ENDFOR
currentTurn ← 0
nobodyHasWon ← True
//Loop until game over
WHILE nobodyHasWon DO
      //If currentTurn at player limit reset to first player
      IF currentTurn MOD numPlayers = 0 THEN
            currentTurn ← 0
      ENDIF
      //Display the latest version of the board to the screen
      OUTPUT DrawCurrentGameBoardState()
      //Display player data
      OUTPUT "It's " + playerNames[currentTurn] "'s turn"
      OUTPUT playerInventories[currentTurn]
      //Get user input required to place a word
      wordInput ← USERINPUT "Enter a word using your tiles"
      coordInput ← USERINPUT "Enter coordinate e.g. A5"
      rotInput ← USERINPUT "Enter 'right' or 'down'"
      //If ValidInputs method returns True for all inputs
      IF ValidInputs(wordInput,coordInput,rotInput) THEN
            //Place the word in specified position on the board
            PlaceWord(wordInput,coordInput,rotInput)
            //Remove the letters used to create the word
            RemoveLetters(wordInput, playerInventories[currentTurn])
            //Distribute tiles to the player as a replacement if there are
            //any available
            DistributeLetters(playerInventories[currentTurn])
            //Calculate word score plus old score
            tempScore ← CalcScore(wordInput) + playerScores[currentTurn]
            //Assign new score to player
            playerScores[currentTurn] ← tempScore
      ELSE
            OUTPUT "Invalid input"
      ENDIF
      //If player still has no tiles after distribution, they have ended the game
      IF playerInventories[currentTurn].Length = 0 THEN
            nobodyHasWon ← True     //Exit game loop
      ENDIF
      //Increment counter to access next player's data
      currentTurn ← currentTurn + 1
ENDWHILE
//Executed on game end, showing leader board of players and scores
OUTPUT DrawLeaderboard(playerScores)
```

Designing game systems

I have decided to split the program into smaller programs that will allow me to prototype the main features of the game. This way I'm able to evaluate if each feature is computationally solvable, as well as being able to resolve problems I may run into, more efficiently, before I merge these modular systems together, where it may become harder to manage.

At the completion of each sub system prototype, I will evaluate whether the system achieves its intended purpose. I will also be testing each system with various inputs and data types, with the intention to break it. From this, I can begin to iterate, including error prevention methods until I produce an independent solution.

Method dependency / Validation considerations

A consideration that must be made regarding the deconstruction of larger systems into smaller systems, as mentioned above, is the validation and dependency or lack-of, shared between each individual system.

During the development of this project, I will be producing many systems, which ideally would each perform validation on the data that is passed to them, this would be done with the goal of producing self-dependent systems, which do not assume the data passed to them is valid. The benefit of this is that if each system is an abstraction of a process, and is self-dependent, it could then be reused in another program/project. However, the amount of time and design required to produce systems in this manner exceed the amount of time allocated for this project.

To reiterate, I do not intend to re-use the majority of systems that are produced in this project, as the systems I will produce won't have a high enough level of abstraction to be used in other applications. The main reason for this is because the aim of this project is to recreate Scrabble, not to focus my efforts on creating a collection of independent systems and libraries that can be reused.

So, to conclude, I will be developing systems that will have some dependence on each-other, meaning that, for instance, function B will assume that the passed parameters are the correct data type, where a method A, has already validated the data, before passing it to function B. As another, more specific example: method A receives a user input which will then be checked against a dictionary, if this returns true, the data is then sent to function B and further processed. In this example "method A" is the only system that will perform the validation, with further functions, like "function B", performing no checks on the data, only processing it.

Test data

Throughout the iterative agile development of this project I will be testing and attempting to break the systems I produce, with the intention to resolve issues before the client receives the product. Usually, for a project similar to this, a set of data, such as a set of user inputs can be defined before development, which would be used to assess the system, determining if it is adequately user-friendly. However, with this project and the randomised nature of Scrabble, I cannot pre-define a set of test data. Instead, I will have to playtest the solution as it is being developed, also allowing users, external to the development process to use the program, as this will most-likely uncover issues I may not encounter, which can then be resolved.

Post development

Once the solution has been fully developed, it will then be time to play the game in its entirety. Reiterating the statement above, due to the randomised distribution of tiles, each game will be different, meaning that many potential avenues for the solution to fail or throw an exception will be explored, and so the final solution can be evaluated.

Development

Sub system – Turn based input

Firstly, the highest level of design for my program must consist of a set of initialisation methods and declarations, setting up the starting conditions, values and states of each system. It should only be executed once. This is followed by a game loop or an "infinite loop", either way its purpose is to repeatedly process data until the program is exited or until a condition(s) is met. Each iterative cycle processes the same set of methods, but with different inputs and new outputs.

Going a few levels deeper, the initialisation methods will include functions and assignments that do not change as the game progresses. For example, on execution, the program must know the number of players that will be interacting with it, the value of this data will not change throughout the game loop. This is vital for the game to know its current state at any point, including knowing whose turn it currently is, the letters each player has, and at what point to cycle back to the first player.

A case where initial values do change is with each player's scores, which by default each will be zero. Yet, will increment by the score gained for each successful word placement.

The aim of this system is allowing the main game loop to access the current / next player's data. Such as the current player's name that has been assigned to them. After displaying the player's information, it is then the next players "turn", and the same process repeats. When the loop reaches the last player defined, the expected result is to reset to the first player, and continue.

Development – Turn based input

Here I have designed a small program that will ask for the number of players as an integer, then using that allow the user to enter a name for each player. The program will infinitely loop, until the program is closed. Displaying the current player's name, then the next, and once it realises that it is looking at the last player, reset to the first.

I could have abstracted this program to be independent of a game, by using less specific variable names by not including the word "player", however, I don't plan to reuse this code and I will be using this program within the final solution.

Code

```
#Methods
def SetupPlayers():
    #Uses global variable numPlayers to determine
    #number of loops
    for x in range(numPlayers):
        newPlayerName = input("Enter player name")
        #Add input to global variable
        playerNames.append(newPlayerName)
#Init
playerNames = []
#Cast input as in integer
numPlayers = int(input("Enter the number of players"))
SetupPlayers()
nobodyHasWon = True

currentTurn = 0
#Game Loop
while nobodyHasWon:
    #Return back to first player index
    if (currentTurn % numPlayers == 0):
        currentTurn = 0
    #Display user data based on current turn
    print("It's " + playerNames[currentTurn] + "'s turn")
    #Waits for input to prevent infinite spam
    input("Tap enter")
    #Next player index
    currentTurn+=1
```

Output

```
Enter the number of players2
Enter player nameLuc
Enter player nameWill
It's Luc's turn
Tap enter
It's Will's turn
Tap enter
It's Luc's turn
Tap enter
It's Will's turn
Tap enter
```

currentTurn has reset to 0.

Sub system breakdown

There are many issues with this code. Firstly, the usage and modification of global variables within the procedure `SetupPlayers`, by using `numPlayers` and `playerNames`, which are both defined outside of the procedure. This is an issue because it removes the modularity that I want to work towards. If I were to make changes or remove the `SetupPlayers` procedure, the program will not work because the global variables are dependent on their assignment within the procedure. To resolve this, I can pass the variables in as parameters of `SetupPlayers`, making it a function, and then return an array for `playerNames` to use instead.

The second major issue is that if the user we were to input the wrong data type the program would crash, without warning. This occurs because I am relying on the user to enter an integer as a string, so that it can be cast it as an integer, but if they enter anything other than an integer, such as a letter, the program will crash because it cannot convert a letter to a number. To resolve this issue, I need to include some verification methods to prevent these strange inputs, also notifying the player that the input is invalid.

Improved code

```
#Methods
def SetupPlayers(numPlayers_in):
    #Temporary local variable
    names=[]
    #Using parameter instead
    for x in range(numPlayers_in):
        newPlayerName = input("Enter player name")
        #Add input to local variable
        names.append(newPlayerName)
    return names
#Init

#Repeatedly ask for an input, until it is valid
while True:
    #Try to convert string to int
    try:
        numPlayers = int(input("Enter the number of players"))
    #Called when input is the wrong data type
    except ValueError:
        print("You must enter a number")
        continue #Continue loop
    break #Exit the infinite loop

playerNames = SetupPlayers(numPlayers)
....
```

Output

```
Enter the number of playersNo
You must enter a number
Enter the number of players1
Enter player nameLuc
It's Luc's turn
Tap enter
It's Luc's turn
Tap enter
```

Exception output message.

Sub system breakdown of improvements

Firstly, I made it so that `playerNames` is equal to the result of the `SetupPlayers` function. Making use of a passed parameter instead of using a global variable, as well as using a temporary local variable to store the resultant `playerNames` array as it is being built.

Then to prevent any invalid data type from being used in place of the number of players, I included some error prevention. Using Pythons built-in try-catch method, within an infinite loop. This means that any code that may throw an exception can be executed safely without the program crashing. If there is no exception the break, statement will cause the program to break out of the infinite loop and continue running. However, if the statement does error due to an invalid data type or (`ValueError`) the error message will be printed and the infinite loop will continue.

Comparison of output

Here is a trace table for the `numPlayers` variable, comparing the original method and the improved method.

User input	Intended output	Old output	Improved output
51	Valid	Valid	Valid
5%	Invalid input	ValueError	You must enter a number
hello	Invalid input	ValueError	You must enter a number
G	Invalid input	ValueError	You must enter a number

Sub system – Distributing and removing tiles

I will be using the system I created above, attaching onto it a system that after defining the number of players will use a finite sized list of tiles to distribute a number of tiles (7) to each player, removing them from the list. Each turn a player can use the tiles available to them to form a word. If the input can be produced with the tiles given, (one tile can only be used once), it will then be announced as valid and the tiles used will be removed from the player's inventory.

Finally, once the tiles have been removed from the current player, a dealer function will distribute the number of tiles required for a full inventory of (7). If there aren't enough tiles to distribute for a full (7 element) inventory, the function should give as many tiles as it can, until the finite sized list of tiles is empty and no more tiles will be given out as replacements.

Development – Initial distribution

Firstly, I developed a program that assigns each player seven random tiles from a `availableTiles` variable, which stores each tile available in the official game of Scrabble. Given the number of players to create inventories for, the function picks a random index position within the size of the `availableTiles` list, it then assigns it to the current player it is creating an inventory for, also removing the tile from the `availableTiles` list. After it has assigned seven tiles to the current player it appends (adds to the end) the inventory to a two-dimension array (an array of arrays). Finally, when then function creates the inventory for the last player it returns the two-dimension array of player inventories back to the main program.

Code

Required module for `randint` function.

```
import random

def SetupLetters(numPlayers_in):
    playerInventoriesTemp=[] #Temp storage for all player inventories (2D)
    for x in range(numPlayers_in): #For each player
        currentPlayerInventory=[] #Temp storage for this player's inventory
        for y in range(7): #For each letter
            #Pick random index of letter within total bag size
            randomIndex = random.randint(0,len(availableTiles)-1) #Inclusive,exclusive
            #Assign current player that random tile
            currentPlayerInventory.append(availableTiles[randomIndex])
            #Remove tile from total bag
            del availableTiles[randomIndex]
        #Store created inventory
        playerInventoriesTemp.append(currentPlayerInventory)
    #Return 2D array of inventories
    return playerInventoriesTemp

availableTiles = [" "," ","E","E"...]
#Test for two players
print(SetupLetters(2))
```

Returns a random value within the range of the given values. Including the first value, but not the last (Inclusive, exclusive)

E.g. (0,2) includes: 0 and 1.

Output

```
[['I', 'T', 'D', 'O', 'E', 'E', 'U'], ['B', 'R', 'N', 'E', 'R', 'P', 'U']]
```

Sub system breakdown

This function provides the expected result, returning an array that contains an array of tiles for each player specified (in the code two were defined). The only issue with the program is that I am modifying a global variable `availableTiles` within the function, meaning that, if the variable did not exist outside of the function or was renamed the program would error. However, the trade-off for modularity here is worth it, as I do not have to consider parsing around the `availableTiles` variable then reassigning it after modifying it. Furthermore, the direct usage of the variable will only be within this function and within functions like the dealer that I still need to develop.

Development – The Dealer

Finally, it is time to create the system that will interface a player's inventory with the `availableTiles` variable. Distributing the correct number of tiles, randomly to the given inventory, for a sum of seven where possible. Where impossible, distribute nothing and announce that there are no tiles left in the bag.

Code

```
import random

def Dealer(playerLetters_in):
    #Old inventory
    print(playerLetters_in)
    #Show current number of tiles in the bag
    print("There are ",len(availableTiles), " letters left")
    #If the bag is not empty
    if (len(availableTiles)>0):
        #Calc number to deal
        numToDeal = 7 - len(playerLetters_in)
        #If not enough in bag to fill up players collection
        #Specify to give all remaining
        if (numToDeal > len(availableTiles)):
            numToDeal = len(availableTiles)
        #Deal amount to player and remove from bag
        for x in range(numToDeal):
            #Pick random index within range
            randomIndex = random.randint(0,len(availableTiles)-1)
            #Add letters to local inventory variable
            playerLetters_in.append(availableTiles[randomIndex])
            #Remove from list
            del availableTiles[randomIndex]
    else:
        print("No tiles left in bag to distribute")
    #Num tiles after distibution
    print("There are now",len(availableTiles), " letters left")
    #Returns new inventory
    return playerLetters_in

#Simulate a players inventory after placing 4 letter word
playerInventory = ["F","R","L"]
availableTiles = [" "," ","E","E","I","N","R","T","B"]

#Test for two players
print(Dealer(playerInventory))
```

del removes a specified object. In this sample, I am removing the randomly picked tile from the "bag" variable.

Output – Enough tiles to distribute fully

```
['F', 'R', 'L']
There are  9  letters left
There are now 5  letters left
['F', 'R', 'L', 'N', 'B', 'T', 'I']
```

Output – Only one tile left

```
['F', 'R', 'L']
There are  1  letters left
There are now 0  letters left
['F', 'R', 'L', 'B']
```

Output – No tiles left

```
['F', 'R', 'L']
There are  0  letters left
No tiles left in bag to distribute
There are now 0  letters left
['F', 'R', 'L']
```

Error message.

Sub system breakdown

For clarity, the outputs are in the order of input player tiles, number of tiles in the "bag" before distribution, number of tiles in the "bag" after distribution, then finally the output inventory.

In the first test, using the variables that are hardcoded in the code sample above, simulating the player's inventory after placing a four-letter word. With this the function returned the old players inventory plus the required number of random tiles for a sum of seven, as there are enough tiles to distribute fully.

The second output displays the function returning the last tile in the "bag", despite it not bringing the players inventory sum to seven. To show this, I hardcoded the `availableTiles` variable to only contain a "B". Which correctly output.

The finally output, shows the response of the function when the "bag" has no tiles that can be distributed. Which is a small message, which will be displayed whenever a player has placed a word and is asking for replacement tiles. The player's final inventory values are unchanged as to tiles can be added, which is correctly output.

Luc Heaton

Development – Removing tiles

After the successful placement of a word, the used tiles must be removed from the player's inventory of available tiles. This could simply be implemented by passing the players inventory as well as the word they have created into a system. Then for each letter within the word placed, remove the first instance of that letter within the player's inventory.

As a further thought, earlier I mentioned the capability to enter a space during word placement, which obviously the player will not have within their inventory, meaning if the current letter being removed is a space, the system should not try to remove the letter, as it does not exist. Furthermore, if a player uses a blank tile to create a letter they do not have, the same issue will occur. Meaning that I must detect if a space is entered or if a blank is used, and not remove them from the player's inventory.

Code

```
#Takes input of a players tiles and the word successfuly placed
def LetterDeductor(playerLetters_in, wordPlaced):
    #For each letter in the word
    for x in wordPlaced:
        #Ignore blank value
        if (x != " "):
            #Grab the current letter index
            letterIndex = playerLetters_in.index(x)
            #Remove current letter from player
            del playerLetters_in[letterIndex]
    #Doesn't require return as we're using object references
    #So the players inventory is directly modified

#Test values
playerInventory = ["H","E","L","L","O"]
placedWord = "HELLO"
print("Before removal:" , playerInventory)
LetterDeductor(playerInventory,placedWord)
print("After removal:" , playerInventory)
```

Output

```
Before removal: ['H', 'E', 'L', 'L', 'O']
After removal: []
```

Sub system breakdown

Firstly, using a set of test values, that represent the format of the data that will be passed into the procedure in the full program, I passed them into the procedure. Looping through each letter within the `wordPlaced` (the successfully placed word). If the current letter is a space, we ignore it and move onto the next value, otherwise we grab the index position that the letter appears at, using the `index` method of a list. It returns the first index position that a value appears at, which I have illustrated below. With the grabbed index position, the same delete method is used to remove the letter from the player's inventory.

This process repeats until the loop has processed the last character within the `wordPlaced` inputted. Once, this happens the program continues to execute from the point it was called, with no value being returned by the procedure. This works because Python does not make a copy of list parameters that are used locally, essentially when a value is passed as a parameter, the procedure does not create a new copy of that parameter that can only be used within that function. Instead, a reference to the object (value) passed in is used, meaning the given value is directly changed by the procedure, and does not require assignment. I have also given an example of this below.

Luc Heaton

Examples

Code – Index usage

```
letters = ["C","A","A","B"]
print("Index equals:" ,letters.index("A"))
```

Output – Index usage

```
Index equals: 1
```

First instance of letter index.

Here you can see index method returns the index of the first occurrence of the value passed.

Code – Procedures

```
#Changes first value to a 4
def Change(values):
    print("Parameter ID:", id(values))
    values[0] = 4

myValues = [1,2,3]
print(myValues ,"with ID", id(myValues))
#Call function
Change(myValues)
#Values after
print(myValues)
```

Gets the variable's identifier.

Output – Procedures

```
[1, 2, 3] with ID 2706041438536
Parameter ID: 2706041438536
[4, 2, 3]
```

Same identifier.

With the simple function, that takes a list as a parameter and changes, the first value to a four. You can see that despite the value being a parameter, Python uses a reference (id value) to the same object. As shown by the fact the local variable (which is not really a local variable, in this case) shares the same reference id as the global variable. Therefore, the procedure does not require a return value.

Sub system further breakdown and development

Even though the system achieves its purpose, it does not consider the usage of a blank tile. This could easily be fixed as we have already resolved the issue of a letter being a space. We can assume that if the player does not have a letter and it's not a space, then the player is using a tile, so instead of removing the letter created by the blank tile, remove a blank tile from the player's inventory (taking the form of a space " ").

Improved code

```
#Takes input of a players tiles and the word successfuly placed
def LetterDeductor(playerLetters_in, wordPlaced):
    #For each letter in the word
    for x in wordPlaced:
        #Ignore blank value
        if (x != " "):
            #Attempt to remove letter
            try:
                #Grab letters index
                letterIndex = playerLetters_in.index(x)
                #Remove from player
                del playerLetters_in[letterIndex]
            #If letter doesnt exist assume blank tile usage
            except:
                #If cant remove letter remove blank tile
                del playerLetters_in[playerLetters_in.index(" ")]
    #Still doesnt require a return

#Test values
playerInventory = ["H","E","L","L"," "]
placedWord = "HELLO" #Using blank for letter O
print("Before removal:" , playerInventory)
LetterDeductor(playerInventory,placedWord)
print("After removal:" , playerInventory)
```

Improved output

```
Before removal: ['H', 'E', 'L', 'L', ' ']
After removal: []
```

Sub system breakdown

I settled for the try-except method, as it was the simplest implementation of testing if the player had a letter, otherwise remove a blank tile from the inventory.

Using a set of test values to simulate placing the word "HELLO", and using a blank for the letter "O", the procedure removes the available tiles, and because the player does not have the letter "O", the blank is removed instead.

Sub system – Array to grid display

The aim of this system is to establish a method(s) of initialising the empty two-dimensional array of a fixed size, displaying it in a text-based format in the layout of a grid. This would act as the main output of the main program, allowing the player to read the grid (game board) and specify the location of their next word placement. Every time a change is made to the board i.e. a new word is added; the board must update accordingly.

Development – Array to grid display

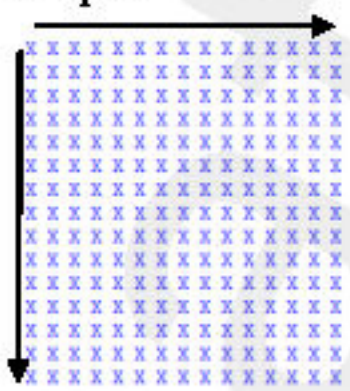
Firstly, creating a 2D list and populating it with placeholder values will be required to display anything at all. Using this data, I must loop through it and upon reaching an index position of the row size, which for a 15x15 grid would be 15 (technically 14 for an array). However, upon reaching this row size the next value of the list must be printed on the next line. Creating a grid layout of values, left to right, top to bottom.

Code

```
def DrawGrid():
    #For each item in board
    for y in range(15):
        for x in range(15):
            if (x % 15 == 0): #If at end of the row
                print("") #Start new line
            #Print current
            print(gameGrid[x][y], end = " ")

#Instantiate array of X's size 15x15
gameGrid=[["X"]*15 for i in range(15)]
#Draw grid using 2D list
DrawGrid()
```

Output



Sub system breakdown

This solution works well, making use of Pythons print arguments, allows the appending of values to the same line. By using the optional argument *end=*, allows you to define what is used after the output, usually this defaults to "\n" which starts a new line. However, changing this to a space allowed me to create a line of items one space apart. Pairing this with a condition to check if the current index is the width of the grid (hardcoded as 15), allows me to now make use of Pythons defaulted new line print, to start a new row.

The method works well, but it depends on a global 2D list variable called *gameGrid*. Furthermore, this will only work for a grid size of 15x15, but for Scrabble this is fine. I may modify it, so that the 2D list to be drawn is specified as a parameter. Allowing this function to be reused by any 15x15 2D list.

Sub system improvements

As another note, this is a very hard display to read. There are not any markings or coordinates that can be read from the axes, to easily specify the placement location of a word. I could implement this by adding numbers and letters before and during drawing the grid. For example, I could print letters A through O before I start drawing the grid as the top row, then for each new row print an incrementing number.

Aims:

-Make the 2D List being drawn, a parameter.
-Display letters on the top row of the grid.
-Display numbers along the side of the grid.

Code

```
def DrawGrid(grid_in):
    #Output top row coords
    print("   A B C D E F G H I J K L M N O", end = "")
    #For each item in board
    for y in range(15):
        for x in range(15):
            if (x % 15 == 0): #If at end of the row
                print("") #Start new line
                print("%2d" % (y+1,), end = " ")
            #Print current
            print(grid_in[x][y], end = " ")

#Instantiate array of X's size 15x15
gameGrid=[["+"]*15 for i in range(15)]
#Draw grid using 2D list
DrawGrid(gameGrid)
```

I mean +'s.

Output

```
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

`["+"]*15` Create fifteen "+" elements, forming: *["+","+"...]*.

`for i in range(15)` Repeat this process fifteen times, forming: *[["+","+"..],["+","+"...]...]*.

Sub system breakdown of improvements

The method of displaying the letters is quite poorly implemented, however, it does create the look I want for the top row of coordinates. The formatting of the side column went through a few iterations though, before I came across a method of formatting the number so that a single digit has a space before it, and double digits do not. As well as changing from X's to +'s, giving a very clean and readable look. Finally, using y+1 instead of just y, so that coordinates can be references A1 through A15 instead of A0 to A14.

Some of the methods I had prior to creating the method above include double digits, which work but are quite hard to read with the un-necessary leading 0's. My first attempt was without any formatting, just using the single/double digit, which caused the grid to be shifted by one character, due to the extra digit for numbers greater than nine.

Formatting Code Used	`print(y, end = " ")`	`print('{:02}'.format(y), end = " ")`

Output	`  A B C D E F G H I J K L M N O`
	`0 + + + + + + + + + + + + + + +`
	`1 + + + + + + + + + + + + + + +`
	`2 + + + + + + + + + + + + + + +`
	`3 + + + + + + + + + + + + + + +`
	`4 + + + + + + + + + + + + + + +`
	`5 + + + + + + + + + + + + + + +`
	`6 + + + + + + + + + + + + + + +`
	`7 + + + + + + + + + + + + + + +`
	`8 + + + + + + + + + + + + + + +`
	`9 + + + + + + + + + + + + + + +`
	`10 + + + + + + + + + + + + + + +`
	`11 + + + + + + + + + + + + + + +`
	`12 + + + + + + + + + + + + + + +`
	`13 + + + + + + + + + + + + + + +`
	`14 + + + + + + + + + + + + + + +`
		`  A B C D E F G H I J K L M N O`
		`00 + + + + + + + + + + + + + + +`
		`01 + + + + + + + + + + + + + + +`
		`02 + + + + + + + + + + + + + + +`
		`03 + + + + + + + + + + + + + + +`
		`04 + + + + + + + + + + + + + + +`
		`05 + + + + + + + + + + + + + + +`
		`06 + + + + + + + + + + + + + + +`
		`07 + + + + + + + + + + + + + + +`
		`08 + + + + + + + + + + + + + + +`
		`09 + + + + + + + + + + + + + + +`
		`10 + + + + + + + + + + + + + + +`
		`11 + + + + + + + + + + + + + + +`
		`12 + + + + + + + + + + + + + + +`
		`13 + + + + + + + + + + + + + + +`
		`14 + + + + + + + + + + + + + + +`

Notes	This method doesn't take into account the extra digit that occurs after 9, Which causes the shift. Also, impacting the placement layout of letters.	This method adds a leading 0 to each single digit number. Fixing the shift issue and the layout of letters. However, it doesn't look very clean and readable.

Sub system evaluation

After getting the correct formatting for the side column numbers, the system works well. It is also modular, as it has no external dependencies on variables. To make it more modular I could calculate the list width and height (given it's a uniform list) from the passed parameter, and use those values instead of the hardcoded 15's. However, for my usage within Scrabble, I do not need to implement this functionality.

Sub system – Grid based word placement interface

Once, I am able to represent the two-dimensional array data as a visual text-based grid, the next step will be inserting strings into the grid using a text interface. Requiring the user to enter a string followed by a set of coordinates in the format "A10" or "D3", as well as a value indicating whether the word is to be placed to the right or downward. After inputting the fields, I must validate the string only contains alphabetic characters and spaces; the coordinate input is within range, does not contain any strange characters and is in the correct formatting. Finally, the orientation must be one value or another; it cannot be anything else (such as a Boolean). If the values are all valid, then using the specified coordinate as a starting point, begin placing each letter in the correct orientation.

Development – Coordinate validation

The first task will involve validating an entered coordinate, which must be in the correct formatting (*LetterNumber*), and must be in the range of A to O, 1 to 14.

I could achieve this by firstly checking if the input is between two and three characters (as letters are single digit and the board size requires up to double digit numbers), if this is true the input string can be split into two substrings; the first character being one and any further characters being the number (XY or XYY). With these two substrings I could validate if the first substring is a letter and within a range of letters (A-O) and if the second substring is an integer and within a range of (1-14).

Code

```
#Checks if input is an integer
def CheckIfInt(value_in):
    try:
        int(value_in)
    except ValueError:
        return False
    return True

def ValidateCoord(coord_in):
    validCoordsX = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O"]
    #If input is two or three digits
    if (len(coord_in) <= 3 and len(coord_in) >= 2):
        #Get substrings
        firstChar = coord_in[0].upper() #Convert to uppercase
        secondChar = coord_in[1:]

        #If first character is within valid coordinates
        if (firstChar in validCoordsX):
            #Check if second value is an integer
            if (CheckIfInt(secondChar)):
                #If within range 1-14
                if (int(secondChar) >= 1 and int(secondChar) <= 14):
                    return True
    return False
```

Output

Inputted coordinate (string)	Data Type	Program Output	Notes (The first causation of error- if any)
A	Erroneous	False	As the input is not within 2 or 3 chars
G5	Typical	True	As the input is 2 digits, the first character is a letter, and the second substring is a number within the range 1-14
O14	Typical	True	
A0	Extreme	False	As the second substring is not within the range 1-14
Z15	Extreme	False	As the first substring is not within the *validCoordsX* list
Ba%	Erroneous	False	As *CheckIfInt* returned false, for the string "a%"

Sub system breakdown

I realised that I may have to validate a string input as an integer at other points throughout my program, so I produced a simple function called `CheckIfInt`, which takes a single value (of any type) as a parameter. Using Pythons try-except methods, I can safely (execute without throwing an error) attempt to cast the value as an integer by using `int`, if this causes a `ValueError` (incorrect data type) then the function returns false. Otherwise, `return True`.

The main function of this sub system is `ValidateCoord`, which takes in a single string parameter. Creates a local string list variable containing all the valid x coordinates (A-C), then executes through a series of validation conditions. Firstly, it checks if the length of the string to be between 2 and 3 characters long, if this is true, the string is split into substrings using Pythons string manipulation ([x:y]), assigning `firstChar` to the first character in uppercase (if possible), then grab everything after the first character for the `secondChar`.

With these values, the first check is if the `firstChar` is a value within the `validCoordsX` list, proving if the character is alphabetic and within the correct range. Then the value is passed into the external method checking if the value can be converted into an integer, the return value of this function determines if the final check will be executed, which validates if the `secondChar` (the y coordinate) is within the range 1-14.

If any of these checks fail, the function will return False, otherwise if all the checks are met, true is returned.

Development – Converting the coordinate to an index

The second, smaller yet fundamental task will be to convert an English formatted coordinate e.g. F11 to an index position that identifies a cell / element within the 2D list that stores the data for the current game board state. For example, the coordinate A1 will represent the 0, 0 index within the board, A2 will be 1, 0 and so on.

To achieve this, the simplest method I can think of is using the same `validCoordsX` variable containing all the valid letters A-O and grabbing the index position of the letter inputted. Meaning, for the first character of the input B6, the letter B will be at index position 1 (list indexing starts at 0). This solves the indexing for the x-axis. For the y-axis, I could simply take the second substring, containing the number minus one. So, for the example B6 the index would be 1, 5.

Code

```
#Takes in a validated coordinated e.g. G12
def ConvertCoordToIndex(coord_in):
    validCoordsX = ["A","B","C"...]
    #Grab substrings again
    firstChar = coord_in[0].upper() #Convert to uppercase
    secondChar = coord_in[1:]
    #Get index position of firstChar within validCoordsX
    firstIndex = validCoordsX.index(firstChar)
    #Cast second substring as integer and take one
    secondIndex = int(secondChar)- 1
    #return indexs in list format
    return [firstIndex,secondIndex]

indexPos = ConvertCoordToIndex("B6")
print(indexPos)
```

Output

```
[1, 5]
```

Sub system breakdown

Working through my written logic, I have produced a function that achieves the same function. Taking in a validated coordinate (which in the full program would be completed by the `ValidateCoord` function), this will then be split into substrings as seen before. Instead of this, I could have produced a separate function that produces these substrings for a coordinates input. However, I probably will not be doing this very specific task any more throughout this project. However, if I do repeat the code another time I will produce a separate method.

Continuing through the function. I made use of Pythons `index` property method, which takes the element being searched for as a parameter, trailing the list being searched, and returning the index position of the element. I can safely do this because I know that the coordinate will be within the list. As the input will be passed into the `ValidateCoord` function first, checking if the coordinate is valid for me.

Finally, as I require both indexes and list indexing is convenient. I returned both indexes within a list. With the x-axis index being the first element, and y-axis index being the second.

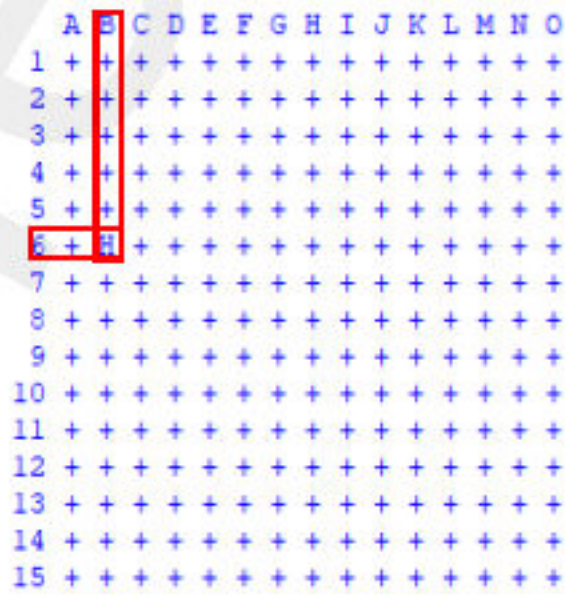
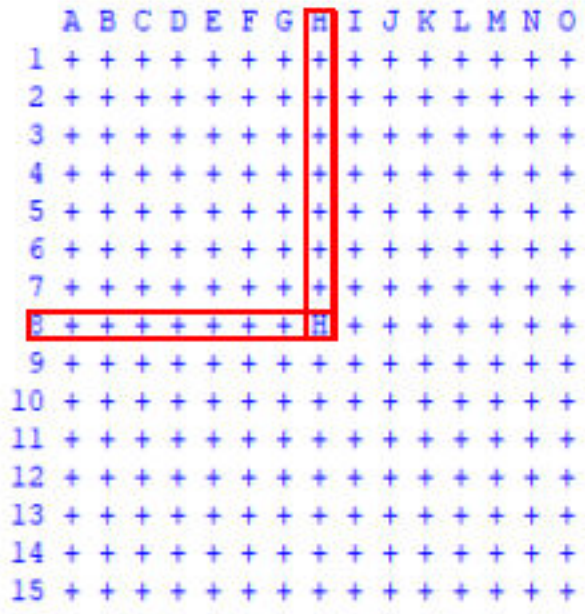
To further, illustrate the method's use. I paired it with the `DrawGrid` function. Assigning the specified coordinate index, a value within the game board. Then re-drawing it.

```
indexPos = ConvertCoordToIndex("B6")
print(indexPos)

gameGrid[indexPos[0]][indexPos[1]] = "H"
DrawGrid(gameGrid)
```

Here I have retrieved the index positions for the coordinate B6, and using these indexes, assigned the letter H, and then re-drawn the grid.

Outputs for different coordinates

Input coordinate	Output index positions	Grid output
B6	[1, 5]	
H8	[7, 7]	

Development – Placing the words

Currently, the only method of accessing cells within the grid is by indexing the 2D array, and modifying elements on a letter-by-letter basis. Meaning, that processing a placement will require a starting position (which will be the inputted coordinate) and using the orientation that will also be provided, loop through the grid and assign each cell the next letter from the word input. For example, the most obvious method would be using the length of the word inputted as the loop condition, so for the word "HELLO", iterate five times, assigning the next index, which would be something like: `currentIndex = startCoordinateIndex + currentIteration`.

This method of accessing elements would work for both vertical and horizontal traversal of the grid. However, two separate processes will be required to account for either horizontal or vertical placement, as horizontal indexing would look something like : `gameGrid[currentIndex][startCoordinateIndexY]` in Python syntax as 2D lists are indexed by their x component then y component, in two separate brackets. With `currentIndex` being the x component (the iterating value) and `startCoordinateIndexY` is the y component (the static value).

Firstly, I am going to produce a method of placing hardcoded words, making use of the iteration idea above.

Code

```
#Given a boolean, string and index position array
#Orientation: True=right False=down
def PlaceWord(orientation,word_in,startPos):

    #Insert validation methods here

    #If placed to the right
    if (not orientation):
        #For each letter in the inputted word
        for x in word_in:
            #Assign current letter to element within gameGrid
            gameGrid[startPos[0]][startPos[1]] = x
            #Increment position by one
            startPos[1]=startPos[1]+1
    #Opposite for downward
    else:
        for x in word_in:
            gameGrid[startPos[0]][startPos[1]] = x
            startPos[0]=startPos[0]+1

#Main
gameGrid=[["+"]*15 for i in range(15)]

PlaceWord(True,"HELLO",[0,0])
```

`startPos` will contain the return integer array from `ConvertCoordToIndex()`, which is the user-inputted coordinate.

Using two separate processes for the two possible orientations (right and downward). I am able to start from the inputted coordinate, in index form. Assigning the current index within the game board to the current letter.

Then incrementing the x coordinate either index or the y, depending on the orientation.

Populate list of size 15x15

Place horizontally

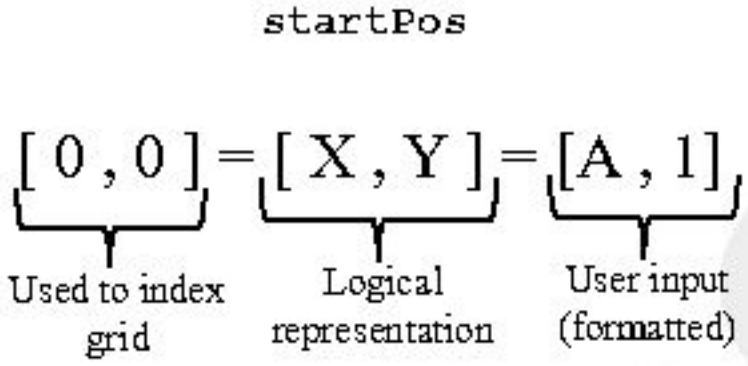
Word to place

Position to place. Where `[0,0]` is equivalent to "A1"

To better illustrate the method I used above, to navigate the grid and assign the correct letters in the expected places, in the correct orientation. I have produced this.

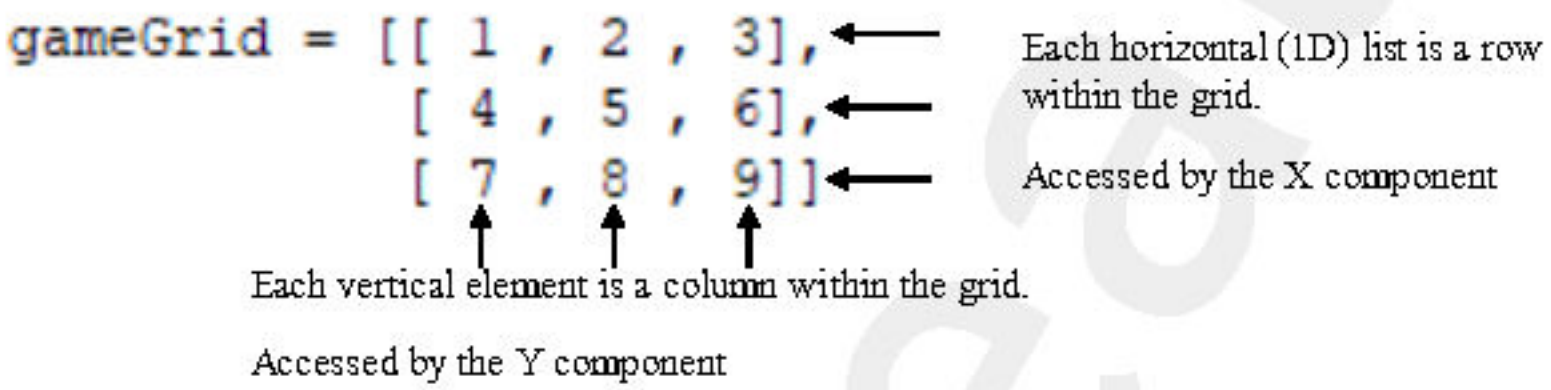
Grid navigation:

Take the coordinate [0,0] as an example, representing the user-input of A1. This is the value of `startPos` within the code.



These values are all equivalent.

Using this I can access the correct position of elements, as the game board list is in this format:



So, by starting with a set of coordinates [0 , 0], I can access the value at 1 within the grid. If I then increment the X component (the left value of the coordinate) by one. I then have access to the first element of the second row which is **4** within the grid (the new coordinate is now [1 , 0]). This can be done for both the X and Y component, allowing horizontal and vertical traversal. In addition, I could instead deduct by one, to access values before a coordinate, allowing backwards travel in either direction.

This is obviously a scaled down version of the grid used within the project, being of size 15x15, not 3x3. To place the word three-letter word "BYE" within this grid, using the method I programmed above, would look something like this:

```
startPos ← [ 0 , 0 ]
gameGrid ← [[ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] , [ 7 , 8 , 9 ]]
wordToPlace ← "BYE"
//Loop for each letter within wordToPlace (loop 3 times)
FOR letter in wordToPlace DO
      //Accessing X/Y components as if they are properties of a class
      //(for simplicity of reading)
      //Assign current letter to current position
      gameGrid[startPos.x][startPox.y] ← letter
      //Increment column position by one
      startPos.y ← startPos.y + 1
```

This pseudocode program would assign the values `[ 1 , 2 , 3]` to `[ "B" , "Y" , "E" ]`, as I'm incrementing the Y component which changes the column position.

However, reviewing the programmed solution. Currently, the method will assign the correct values at the correct positions and orientation - logically. As, executing this code sample will not provide an output at all to represent this update in game state. To fix this I can incorporate the `DrawGrid` function I developed earlier, passing it the updated `gameGrid` variable as a parameter, should display the word "HELLO" in the expected position, horizontally.

Updated code

```
#Given a boolean, string and index position array
#Orientation: True=right False=down
def PlaceWord(orientation,word_in,startPos):

    #Insert validation methods here

    #If placed to the right
    if (not orientation):
        #For each letter in the inputted word
        for x in word_in:
            #Assign current letter to element within gameGrid
            gameGrid[startPos[0]][startPos[1]] = x
            #Increment position by one
            startPos[1]=startPos[1]+1
    #Opposite for downward
    else:
        for x in word_in:
            gameGrid[startPos[0]][startPos[1]] = x
            startPos[0]=startPos[0]+1
    #Update grid
    DrawGrid(gameGrid)

#Main
gameGrid=[["+"]*15 for i in range(15)]

PlaceWord(True,"HELLO",[0,0])
```

Excluded from this code sample is the `DrawGrid` method, which I have already described and shown above.

Space for validation.

The modification to the function.

Output for different words and coordinates

Hardcoded input values	Notes	Grid output
`PlaceWord(True,"HELLO",[0,0])`	True: -Place horizontally. "HELLO": -Word to place. [0,0]: -Coordinate to start from. -Equal to A1	A B C D E F G H I J K L M N O 1 H E L L O + + + + + + + + + + 2 + + + + + + + + + + + + + + + 3 + + + + + + + + + + + + + + + 4 + + + + + + + + + + + + + + + 5 + + + + + + + + + + + + + + + 6 + + + + + + + + + + + + + + + 7 + + + + + + + + + + + + + + + 8 + + + + + + + + + + + + + + + 9 + + + + + + + + + + + + + + + 10 + + + + + + + + + + + + + + + 11 + + + + + + + + + + + + + + + 12 + + + + + + + + + + + + + + + 13 + + + + + + + + + + + + + + + 14 + + + + + + + + + + + + + + + 15 + + + + + + + + + + + + + + +
`PlaceWord(False,"LUC",[6,6])`	False: -Place vertically. "LUC": -Word to place. [6,6]: -Coordinate to start from. -Equal to G7	A B C D E F G H I J K L M N O 1 + + + + + + + + + + + + + + + 2 + + + + + + + + + + + + + + + 3 + + + + + + + + + + + + + + + 4 + + + + + + + + + + + + + + + 5 + + + + + + + + + + + + + + + 6 + + + + + + + + + + + + + + + 7 + + + + + + L + + + + + + + + 8 + + + + + + U + + + + + + + + 9 + + + + + + C + + + + + + + + 10 + + + + + + + + + + + + + + + 11 + + + + + + + + + + + + + + + 12 + + + + + + + + + + + + + + + 13 + + + + + + + + + + + + + + + 14 + + + + + + + + + + + + + + + 15 + + + + + + + + + + + + + + +

LUC Heaton

Sub system breakdown

In future, I must build onto this method with some validation, as in this current state many assumptions are being made, including: the assumption that placement doesn't overwrite any tiles, that the word is of correct length and fits within the board, that a word placement isn't exiting the game board. For example, the word "HI" being placed to the right starting at O1 would cause an out of range error, causing the game to break.

This cannot be acceptable within the final solution, and must be accounted for. Furthermore, in this current state, multiple incompatible words could be placed next to each other, as the resultant words that may be formed during a placement are not being validated.

Following the Gantt chart, I produced, I have reached the halfway point through developing all the sub systems. It would be useful at this point to summarise each method that has been produced so far, stating its inputs, outputs and function.

Method Summary

Method name	Parameters	Usage
SetupPlayers(int)	int - Number of players	Retrieves a name for each player, returning a list of player names.
SetupLetters(int)	int - Number of players	Initialises a list of player inventories with random tiles, which is returned and used throughout the game.
Dealer(char list)	list - Player inventory	Distribute tiles to the passed inventory for a sum of seven (if possible). Returns appended inventory.
LetterDeductor(char list,string)	list - Player inventory string - Word to remove	Remove the tiles used to form the string passed in from the player inventory passed in. Does not return a value.
DrawGrid(2D list)	list - List to draw	Uses the 2D list given to draw a grid, with coordinates on each axis.
CheckIfInt(value)	value - Any type	Returns true if the given value is an integer. Otherwise, false.
ValidateCoord(string)	string - English coordinates	Validate the given string, outputting true if the coordinate is valid.
ConvertCoordToIndex(string)	string - English coordinates	Using the validated English coordinate, convert it to an index-based coordinate, outputted as a two-element list.
PlaceWord(bool,string,int list)	bool - Orientation string - Word to place list - Start position coordinate	Loop through each letter within the word to place, and place it within the grid, from the given starting position, in the direction specified.

To further test the program in its current state, I am going to combine the major sub systems together. With the result of a prototype, allowing me to rapidly test the placement of words within the grid, in a similar method to the final solution. To achieve this, I am going to be making use of all the systems I have created so far.

Prototype – Turn based word placement

To produce this prototype, I am going to slowly implement individual systems, building towards a full prototype over time. Therefore, to begin with, I combined the two main initialisation methods: `SetupPlayers` and `SetupLetters`, as well as the simple game loop I used when producing the `SetupPlayers` method. This simply involved creating a new program containing the two methods as well as any other dependencies they may require (in this case using `random`, so I can use `randint`). Then creating all the global variables that are required, such as `availableTiles` and then finally, calling the two methods, storing their output. For display purposes, I have added an output to display each player's inventory during their turn, showing the systems working together so far.

Code

```
import random
#Methods
def SetupPlayers(numPlayers_in):
    #Temporary local variable
    names=[]
    #Using parameter instead
    for x in range(numPlayers_in):
        newPlayerName = input("Enter player name")
        #Add input to local variable
        names.append(newPlayerName)
    return names

def SetupLetters(numPlayers_in):
    playerInventoriesTemp=[] #Temp storage for all player inventories (2D)
    for x in range(numPlayers_in): #For each player
        currentPlayerInventory=[] #Temp storage for this player's inventory
        for y in range(7): #For each letter
            #Pick random index of letter within total bag size
            #Inclusive,exclusive
            randomIndex = random.randint(0,len(availableTiles)-1)
            #Assign current player that random tile
            currentPlayerInventory.append(availableTiles[randomIndex])
            #Remove tile from total bag
            del availableTiles[randomIndex]
        #Store created inventory
        playerInventoriesTemp.append(currentPlayerInventory)
    #Return 2D array of inventories
    return playerInventoriesTemp

#Init
#Repeatedly ask for an input, until it is valid
while True:
    #Try to convert string to int
    try:
        numPlayers = int(input("Enter the number of players"))
    #Called when input is the wrong data type
    except ValueError:
        print("You must enter a number")
        continue #Continue loop
    break #Exit the infinite loop

availableTiles = [" "," ","E","E","E","E","E","E","E","E","E","E"...]
playerNames = SetupPlayers(numPlayers)
playerInventories = SetupLetters(numPlayers)
nobodyHasWon = True

currentTurn = 0


#Game Loop
while nobodyHasWon:
    #Return back to first player index
    if (currentTurn % numPlayers == 0):
        currentTurn = 0
    #Display user data based on current turn
    print("It's " + playerNames[currentTurn] + "'s turn")
    #Display current users inventory
    print(playerInventories[currentTurn])
    #Waits for input to prevent infinite spam
    input("Tap enter")
    #Next player index
    currentTurn+=1
```

During the process of building this prototype, I am not going to be showing the sub systems (functions / procedures) within the code sample, as it is the same code I have documented and developed above, and it takes up too much space.

I will apply this in future code samples, only showing code that is relevant or that has changed.

I will be hiding code above this point, to simplify each code sample.

I do not like this code block within the main program, so I am going move it to a function.

Calling and storing data from the above two methods.

Displays the current player's inventory of tiles.

Output

```
Enter the number of players2
Enter player nameLuc
Enter player nameWill
It's Luc's turn
['S', 'I', 'O', 'E', 'S', 'N', 'I']
Tap enter
It's Will's turn
['T', 'I', 'T', 'I', 'Y', 'I', 'E']
Tap enter
It's Luc's turn
['S', 'I', 'O', 'E', 'S', 'N', 'I']
Tap enter
It's Will's turn
['T', 'I', 'T', 'I', 'Y', 'I', 'E']
Tap enter
```

System breakdown

Here you can see these first two systems working alongside each other. Firstly, initialising the number of players, and using that data to gather player names as well as generating inventories for both players.

Once the execution reaches the game loop, the current players name is displayed as well as their inventory.

With the next iteration towards the full prototype, I am going to create a small function that gets the number of players, within the range one to four (inclusive), replacing the code within the initialisation. In addition to this, I'm going to incorporate a temporary method of: entering the word to be placed, a value representing a right or downward placement, and finally, making use of the system I developed for coordinate validation (`ValidateCoord()`).

The simplest method of implementing user input for the word to be placed and the orientation would be to apply no validation at all, and assume the input(s) will not break the program. This is fine for now, as I am the only user of the program, and so, to save me some time I am going to simply take user input as the word. The implication of this (despite it being temporary) is that a user could enter a word they don't have the tiles for, then a system like `LetterDeductor()`, would break, causing the execution to be suspended, as it tries to remove (tiles) elements from a (inventory) list that doesn't exist.

Code

```
def GetNumPlayers():
    #Repeatedly ask for an input, until it is valid
    while True:
        #Try to convert string to int
        try:
            numPlayers = int(input("Enter the number of players"))
        #Called when input is the wrong data type
        except ValueError:
            print("You must enter a number")
            continue #Continue loop
        break #Exit the infinite loop
    return numPlayers
```

This method asks "infinitely", until it is given an integer. I must include a maximum and minimum valid integer to this function, as currently there can be a negative number of players, or even 2^{31}-1 players.

```
...
#Init
availableTiles = [" "," ","E","E","E","E","E","E"...]
numPlayers = GetNumPlayers()
playerNames = SetupPlayers(numPlayers)
playerInventories = SetupLetters(numPlayers)

nobodyHasWon = True
currentTurn = 0

#Game Loop
while nobodyHasWon:
    #Return back to first player index
    if (currentTurn % numPlayers == 0):
        currentTurn = 0
    #Display user data based on current turn
    print("It's " + playerNames[currentTurn] + "'s turn")
    #Display current users inventory
    print(playerInventories[currentTurn])
    #Temp get word input
    validWord = input("Enter the word to place")
    #Temp oreintation
    orientation = input("Enter: 1 for right, anything else for downward")
    if (orientation == "1"):
        orientation=True
    else:
        orientation=False
    #Temp coordinate input
    while True:
        validCoord = input("Enter a coordinate e.g A5")
        #If coordinate is valid
        if (ValidateCoord(validCoord)):
            break
        else:
            print("Invalid coordinate")
    indexedCoord = ConvertCoordToIndex(validCoord)
    print("Word: {}, To the right? {}, At positon: {}".format(validWord,
                                                              orientation,
                                                              indexedCoord))

    #Waits for input to prevent infinite spam
    input("Tap enter")
    #Next player index
    currentTurn+=1
```

Other methods that have been copied into the code.

New method call.

Temporary method to retrieving the word to place, orientation and validated starting coordinate.

Very useful string formatting technique, that uses the values of the variables within `format` to populate the `{}` elements within the string in order. E.g. the first set of brackets will be replaced with the first variable within `format`.

This code will now allow an input of a string (the word to place), which is not currently validated, a Boolean (orientation, right or down) and finally an English based coordinated, which is validated using the `ValidateCoord` method, then outputted in its converted index-based format using `ConvertCoordToIndex`.

Output

```
Enter the number of playersk
You must enter a number
Enter the number of players2
Enter player nameLuc
Enter player namePlayer1
It's Luc's turn
['E', 'T', 'L', 'O', 'G', 'K', 'B']
Enter the word to placeLOG
Enter: 1 for right, anything else for downward1
Enter a coordinate e.g A5z11
Invalid coordinate
Enter a coordinate e.g A5g8
Word: LOG, To the right? True, At positon: [6, 7]
Tap enter
It's Player1's turn
['O', 'E', 'S', 'T', 'U', 'I', 'I']
Enter the word to place
```

`GetNumPlayers`, `ValueError` executed because of the invalid type of string.

`SetupPlayers`.

Within game loop.

Invalid as it is out of range.

English coordinate G8 in index form.

As you can see, the prototype now allows a very basic interface that I can use to test the program. The last few features that must be implemented for this prototype to be complete include the `LetterDeductor`, `PlaceWord` and `DrawGrid`. To start with, I've realised the game loop is going to quickly grow in size as more features are implemented, for example, just by adding the coordinate validation and conversion system(s), I've had to include a loop as well as some conditionals within the main game loop. If I continue to append systems in this manner, the code will become increasingly difficult to manage, and that is not even accounting for the issues that may arise if I encounter a problem and have to debug this growing block of code.

A solution to this problem is to encapsulate a large portion of the method calls into one large interface that will manage all the smaller systems, essentially creating a factory method (loosely following the Factory Pattern). Meaning that the game loop would make a call to this factory method, which then manages all the system calls, instead of them being merged with unrelated code, such as updating the current turn and displaying the player's inventory.

In addition, to this I will be implementing the `LetterDeductor` system, which will be removing the tiles used by a player to form a word, if it was successfully placed (this validation has yet to be developed), the tiles will then be removed from the correct player's inventory. To reiterate, in this current state the word being entered is not being validated, and so entering a word that is comprised of tiles that are not available will cause the program to break, but as I have mentioned before, I'm the only user of this program currently, and I'm developing this prototype to most importantly, evaluate the grid placement. The word validation and surrounding word checks will come later.

Luc Heaton

Code

```
...
#The factory method to call other methods
def ProcessTurn(playerLetters_in):
    #Temp get word input
    validWord = input("Enter the word to place")
    ...
    #Remove used tiles
    LetterDeductor(playerLetters_in,validWord)
    print(playerLetters_in)

#Game Loop
while nobodyHasWon:
    #Return back to first player index
    if (currentTurn % numPlayers == 0):
        currentTurn = 0
    #Display user data based on current turn
    print("It's " + playerNames[currentTurn] + "'s turn")
    #Display current users inventory
    print(playerInventories[currentTurn])
    #Call factory manager method, passing current players inventory
    ProcessTurn(playerInventories[currentTurn])
    #Waits for input to prevent infinite spam
    input("Tap enter")
    #Next player index
    currentTurn+=1
```

Rest of processing for inputs.

Removes tiles used to create word, outputs new inventory.

Reduced size of game loop by calling factory method (*ProcessTurn*), passing the current players inventory to be used.

Output

```
Enter the number of players2
Enter player nameLuc
Enter player nameTest
It's Luc's turn
['R', 'I', 'Y', 'Y', 'I', 'F', 'R']
Enter the word to placefir
Enter: 1 for right, anything else for downward2
Enter a coordinate e.g A5A1
Word: fir, To the right? False, At positon: [0, 0]
Traceback (most recent call last):
  File "H:\Py\Scrabble\Documentation\
  Sub Systems\WordPlacingPrototype.py", line 93, in LetterDeductor
    letterIndex = playerLetters_in.index(x)
ValueError: 'f' is not in list
```

Lowercase input causing error.

First letter encountered while looping through the word, which is lowercase.

I quickly came across a small bug in my code, causing a *ValueError*, due to the lack of validation for the temporary word input method. If I enter any lowercase letters for the word as the input, *LetterDeductor* will attempt to remove characters that do not exist, due to the player inventory being a store of uppercase characters. This causes an issue because of the method in which a computer recognises which key has been pressed, by using a unique binary value to represent each character; in our case, the UK uses Extended ASCII, which represents the uppercase character "A" as C1 in hexadecimal and lowercase character "a" as E1. As a quick and simple resolution of this issue, I converted the value of *wordPlaced* to uppercase, within the *LetterDeductor* procedure.

Modified code

```
#Takes input of a players tiles and the word successfuly placed
def LetterDeductor(playerLetters_in, wordPlaced):
    #Convert to uppercase
    wordPlaced = wordPlaced.upper()

    ...
```

The *upper* property method returns a version of the string given in full uppercase. This is a safe implementation, as the method has its own error handling methods, that account for integers and symbols that may be passed, meaning errors that occur within the method will be handled for me, for example the capitalisation of a number, which obviously cannot be done.

New output

```
Enter the number of players2
Enter player nameLuc
Enter player nameWill
It's Luc's turn
['I', 'E', 'J', 'O', 'Y', 'I', 'T']
Enter the word to placetoy
Enter: 1 for right, anything else for downward1
Enter a coordinate e.g A5A1
Word: toy, To the right? True, At positon: [0, 0]
['I', 'E', 'J', 'I']
Tap enter
It's Will's turn
['N', 'E', 'E', 'E', 'T', 'R', 'L']
Enter the word to placeTeN
Enter: 1 for right, anything else for downward
Enter a coordinate e.g A5b2
Word: TeN, To the right? False, At positon: [1, 1]
['E', 'E', 'R', 'L']
Tap enter
It's Luc's turn
['I', 'E', 'J', 'I']
Enter the word to place
```

No longer causes *ValueError*, when using lowercase.

As I am passing the coordinate through a validation method (*ValidateCoord*), I do not need to make any adjustments for it to allow lowercase input. Once, I develop the same system for word input I can remove the fix I just made to the *LetterDeductor* method.

Formatted user-inputs, which will be passed into *PlaceWord* for grid placement.

Retains inventory.

Using the above output as an example, it can be noted that the capitalisation within the user-input, for the word-to-be-placed has no effect on the program, whereas previously the comparison of strings, executed by the `LetterDeductor` would be case sensitive. Which caused an error, whenever, for example, the character "t" from the word "toy" is forcefully removed from a list containing only capitalised characters.

With this small issue resolved, the last implementation of grid-based placement can be added. This should be simple as all the parameters required to interact with the `PlaceWord` method have already be included within the prototype. So, all that is required to add to this iteration is the `DrawGrid` method, which will display the latest state of the game board before the players turn. The `Dealer` to re-distribute tiles after a player has used their tiles, and finally the `PlaceWord` method will be passed all the user-inputs at the end of the turn, which will assign the word in its correct position within the grid.

After, simply copying the methods for `DrawGrid`,`PlaceWord` and `Dealer` into the program, the first modification was made to the `ProcessTurn` procedure, to instead assign the word to its specified position, rather than displaying it. In addition, the addition of a call to the Dealer method, which will attempt to refill the players inventory, given there are tiles available to be distributed. Furthermore, the initialisation of the grid is now required, otherwise there will be no grid to display (this code will be within the main program, outside of any other methods).

Old code

```
def ProcessTurn(playerLetters_in):
    #Temp get word input
    validWord = input("Enter the word to place")

    #Pass validWord to validation method here

    #Temp oreintation
    orientation = input("Enter: 1 for right, anything else for downward")
    if (orientation == "1"):
        orientation=True
    else:
        orientation=False
    #Temp coordinate input
    while True:
        validCoord = input("Enter a coordinate e.g A5")
        #If coordinate is valid
        if (ValidateCoord(validCoord)):
            break
        else:
            print("Invalid coordinate")
    indexedCoord = ConvertCoordToIndex(validCoord)
    #Temp output
    print("Word: {}, To the right? {}, At position: {}".format(validWord,
                                                               orientation,
                                                               indexCoord))

    #Remove used tiles
    LetterDeductor(playerLetters_in,validWord)
```

Old temporary method of displaying the user-input.

New code

```
def ProcessTurn(playerLetters_in):
    #Temp get word input
    validWord = input("Enter the word to place")

    #Pass validWord to validation method here

    #Temp oreintation
    orientation = input("Enter: 1 for right, anything else for downward")
    if (orientation == "1"):
        orientation=True
    else:
        orientation=False
    #Temp coordinate input
    while True:
        validCoord = input("Enter a coordinate e.g A5")
        #If coordinate is valid
        if (ValidateCoord(validCoord)):
            break
        else:
            print("Invalid coordinate")
    indexedCoord = ConvertCoordToIndex(validCoord)
    #Write to grid
    PlaceWord(orientation,validWord,indexedCoord)
    #Remove used tiles
    LetterDeductor(playerLetters_in,validWord)
    #Re-distribute tiles
    Dealer(playerLetters_in)
```

Replacement method, passing user-input into procedure to assign the word to the correct position.

Attempt to fill the player's inventory (sum of 7) with replacement tiles.

Within the `PlaceWord` procedure, a call is made to `DrawGrid`, meaning no further additions to the code are required, as after the word has been assigned to its position, the latest state of the grid will be drawn. The only slight problem with this is that, on the first players turn, the player will not see the state grid until after they have attempted to place a word, as I mentioned above. This is an issue because, for a user that has never used this program before (despite the fact, I am the only current user), the user will not be able to visualise where their word will be placed, let alone the coordinates they must enter to position the word.

As a quick resolution to this issue, a method call to `DrawGrid` could be made after the initialisation of the 2D list used as the game board.

Code

A section from the main program, outside of any other methods.

```
...

numPlayers = GetNumPlayers()
playerNames = SetupPlayers(numPlayers)
playerInventories = SetupLetters(numPlayers)

gameGrid=[["+"]*15 for i in range(15)]
DrawGrid(gameGrid)

nobodyHasWon = True
currentTurn = 0

...
```

Initialise 2D list of size 15x15, storing the current state of the game board.

Draw the starting state of the grid, for player one.

Full Prototype – Turn based word placement

Code

```
#Modules
import random

#Methods

#Retrieve a valid number of players from the user
def GetNumPlayers():
    #Repeatedly ask for an input, until it is valid
    while True:
        #Try to convert string to int
        try:
            numPlayers = int(input("Enter the number of players"))
        #Called when input is the wrong data type
        except ValueError:
            print("You must enter a number")
            continue #Continue loop
        break #Exit the infinite loop
    return numPlayers

#Gather a list of player names to be referenced each turn
def SetupPlayers(numPlayers_in):
    #Temporary local variable
    names=[]
    #Using parameter instead
    for x in range(numPlayers_in):
        newPlayerName = input("Enter player name")
        #Add input to local variable
        names.append(newPlayerName)
    return names

#Distribute 7 random tiles from the bag to each player,
#returning a 2D array containing each players starting inventory
def SetupLetters(numPlayers_in):
    playerInventoriesTemp=[] #Temp storage for all player inventories (2D)
    for x in range(numPlayers_in): #For each player
        currentPlayerInventory=[] #Temp storage for this player's inventory
        for y in range(7): #For each letter
            #Pick random index of letter within total bag size
            #Inclusive,exclusive
            randomIndex = random.randint(0,len(availableTiles)-1)
            #Assign current player that random tile
            currentPlayerInventory.append(availableTiles[randomIndex])
            #Remove tile from total bag
            del availableTiles[randomIndex]
        #Store created inventory
        playerInventoriesTemp.append(currentPlayerInventory)
    #Return 2D array of inventories
    return playerInventoriesTemp

#Checks if a value is an integer
def CheckIfInt(value_in):
    try:
        int(value_in)
    except ValueError:
        return False
    return True
```

```
#Checks if a passed input is in the correct formatting e.g B9
#of correct type and is within the specified range for both axis
def ValidateCoord(coord_in):
    validCoordsX = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O"]
    #If input is two or three digits
    if (len(coord_in) <= 3 and len(coord_in) >= 2):
        #Get substrings
        firstChar = coord_in[0].upper() #Convert to uppercase
        secondChar = coord_in[1:]

        #If first character is within valid coordinates
        if (firstChar in validCoordsX):
            #Check if second value is an integer
            if (CheckIfInt(secondChar)):
                #If within range 1-14
                if (int(secondChar) >= 1 and int(secondChar) <= 14):
                    return True
    return False

#Takes in a validated coordinated e.g. G12
#and convert it to it's equivilent index based value
#e.g A1 = [0,0]
def ConvertCoordToIndex(coord_in):
    validCoordsX = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O"]
    #Grab substrings again
    firstChar = coord_in[0].upper() #Convert to uppercase
    secondChar = coord_in[1:]
    #Get index position of firstChar within validCoordsX
    firstIndex = validCoordsX.index(firstChar)
    #Cast second substring as integer and take one
    secondIndex = int(secondChar)- 1
    #return indexs in list format
    return [firstIndex,secondIndex]

#Takes input of a players inventory and the word successfuly placed
#Removing the tiles used to form the word from the inventory
def LetterDeductor(playerLetters_in, wordPlaced):
    #Convert to uppercase
    wordPlaced = wordPlaced.upper()
    #For each letter in the word
    for x in wordPlaced:
        #Ignore blank value
        if (x != " "):
            #Attempt to remove letter
            try:
                #Grab letters index
                letterIndex = playerLetters_in.index(x)
    #The factory method used to call other methods,
    #removing calls from the main game loop
    def ProcessTurn(playerLetters_in):
        #Temp get word input
        validWord = input("Enter the word to place")

        #Pass validWord to validation method here

        #Temp oreintation
        orientation = input("Enter: 1 for right, anything else for downward")
        if (orientation == "1"):
            orientation=True
        else:
            orientation=False
        #Temp coordinate input
        while True:
            validCoord = input("Enter a coordinate e.g A5")
            #If coordinate is valid
            if (ValidateCoord(validCoord)):
                break
            else:
                print("Invalid coordinate")
        indexedCoord = ConvertCoordToIndex(validCoord)
        #Write to grid
        PlaceWord(orientation,validWord,indexedCoord)
        #Remove used tiles
        LetterDeductor(playerLetters_in,validWord)
        #Re-distribute tiles
        Dealer(playerLetters_in)

#Using the given 2D array, draw it as a grid
#Displays the current state of the grid
#Assumes the grid is 15x15
def DrawGrid(grid_in):
    #Output top row coords
    print("   A B C D E F G H I J K L M N O", end = "")
    #For each item in board
    for y in range(15):
        for x in range(15):
            if (x % 15 == 0): #If at end of the row
                print("") #Start new line
                print("%2d" % (y+1,), end = " ")
            #Print current
            print(grid_in[x][y], end = " ")
    print("") #Start new line
```

Luc Heaton

```
#Given a boolean, string and index position array
#Orientation: True=right False=down
#Assign a word to it's specified position within the 2D array
def PlaceWord(orientation,word_in,startPos):

    #Insert validation methods here
    #Temp capitalise
    word_in=word_in.upper()
    #If placed to the right
    if (not orientation):
        #For each letter in the inputted word
        for x in word_in:
            #Assign current letter to element within gameGrid
            gameGrid[startPos[0]][startPos[1]] = x
            #Increment position by one
            startPos[1]=startPos[1]+1
    #Opposite for downward
    else:
        for x in word_in:
            gameGrid[startPos[0]][startPos[1]] = x
            startPos[0]=startPos[0]+1
    #Update grid
    DrawGrid(gameGrid)

#Called after LetterDeductor to re-distibute tiles
#for a sum of 7 if tiles are available
def Dealer(playerLetters_in):
    #Old inventory
    print(playerLetters_in)
    #Show current number of tiles in the bag
    print("There are ",len(availableTiles), " letters left")
    #If the bag is not empty
    if (len(availableTiles)>0):
        #Calc number to deal
        numToDeal = 7 - len(playerLetters_in)
        #If not enough in bag to fill up players collection
        #Specify to give all remaining
        if (numToDeal > len(availableTiles)):
            numToDeal = len(availableTiles)
        #Deal amount to player and remove from bag
        for x in range(numToDeal):
            #Pick random index within range
            randomIndex = random.randint(0,len(availableTiles)-1)
            #Add letters to local inventory variable
            playerLetters_in.append(availableTiles[randomIndex])
            #Remove from list
            del availableTiles[randomIndex]
    else:
        print("No tiles left in bag to distribute")
    #Num tiles after distibution
    print("There are now",len(availableTiles), " letters left")
    #Returns new inventory
    return playerLetters_in


#Main program
#Init
availableTiles = [" "," ","E","E","E","E","E","E","E","E","E","E","E","E",
                "I","I","I","I","I","I","I","I","I","I","I","I",
                "O","O","O","O","O","O","O","O","N","N","N","N","N","N","R","R","R","R","R","R",
                "T","T","T","T","T","T","L","L","L","L","S","S","S","S",
                "U","U","U","U","D","D","D","D","G","G","G","B","B","C","C",
                "M","M","P","P","F","F","H","H","V","V","W","W","Y","Y","K","J","X","Q","Z"]

numPlayers = GetNumPlayers()
playerNames = SetupPlayers(numPlayers)
playerInventories = SetupLetters(numPlayers)

gameGrid=[["+"]*15 for i in range(15)]
DrawGrid(gameGrid)

nobodyHasWon = True
currentTurn = 0

#Game Loop
while nobodyHasWon:
    #Return back to first player index
    if (currentTurn % numPlayers == 0):
        currentTurn = 0
    #Display user data based on current turn
    print("It's " + playerNames[currentTurn] + "'s turn")
    #Display current users inventory
    print(playerInventories[currentTurn])
    #Call factory manager method, passing current players inventory
    ProcessTurn(playerInventories[currentTurn])
    #Waits for input to prevent infinite spam
    input("Tap enter")
    #Next player index
    currentTurn+=1
```

Luc Heaton

Output

```
Enter the number of players2
Enter player nameLuc
Enter player nameWill
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
It's Luc's turn
['L', 'E', 'D', 'N', 'N', 'I', 'I']
Enter the word to placedine
Enter: 1 for right, anything else for downward1
Enter a coordinate e.g A5f2
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + D I N E + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
['L', 'N', 'I']
There are  80  letters left
There are now 76  letters left
Tap enter
It's Will's turn
['K', 'E', 'Y', 'E', 'I', 'M', 'I']
Enter the word to placekey
Enter: 1 for right, anything else for downward2
Enter a coordinate e.g A5o1
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + K
 2 + + + + + D I N E + + + + + E
 3 + + + + + + + + + + + + + + Y
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
['E', 'I', 'M', 'I']
There are  76  letters left
There are now 73  letters left
Tap enter
It's Luc's turn
['L', 'N', 'I', 'W', 'S', 'T', 'Z']
Enter the word to placel nt
Enter: 1 for right, anything else for downward2
Enter a coordinate e.g A5g1
   A B C D E F G H I J K L M N O
 1 + + + + + + L + + + + + + + K
 2 + + + + + D   N E + + + + + E
 3 + + + + + + N + + + + + + + Y
 4 + + + + + + T + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
['I', 'W', 'S', 'Z']
There are  73  letters left
There are now 70  letters left
Tap enter
```

Within `GetNumPlayers`, currently it is possible to have any number of players, including negatives. This will be limited to a range of 1-4 in the final program.

Within `SetupPlayers`, using the inputted integer to gather the correct amount of player names to be referenced each

`DrawGrid`, to display the starting game state to the first player. Without this initial call to `DrawGrid` the player would not have a clear visualisation of the playing board.

The first iteration of the game loop, accessing the first players name and inventory. Which then makes a call to the `ProcessTurn` procedure.

Currently, not utilising any validation for the word input, just assuming the input is valid, until the validation methods are developed.

Also, using a temporary method for retrieving a Boolean, if the input is "1" then the Boolean is true (meaning a rightward placement), for any-other value (even a return) the Boolean is false.

The only input that is currently fully developed, and validated is coordinate input. Confirming, that the input is of the correct type, in the correct format and within range of the playing board. If not, the program will request another input.

The players inventory after the characters used to form their word have been removed by the `LetterDeductor`.

The number of tiles before and after they are deducted by the `LetterDeductor`, from the `availableTiles` variable

`ProcessTurn` has finished its execution, the `currentTurn` variable has incremented by one, and so the second iteration of the game loop is started now accessing the second player's data.

Entered a value that is not "1" and so the orientation will be downward.

After each placement, the `DrawGrid` procedure is called, drawing the latest game state. Showing the latest placement of "KEY".

In the final program, I plan to have users enter a space character, where an existing tile on the board will be used. In this short test. I attempted to use the existing tile "T", from the word "DINE", within the word "L[I]NT". However, instead of skipping a cell when the program tries to place the space character. It instead overwrites the tile with the space. This will be resolved in future builds.

Prototype evaluation – Turn based word placement

After, the several iterations of implementation for all the methods I have developed thus far, I now have a full prototype, that allows a number of players to place words using the tiles they possess, in a turn-based manner. In its current state, I can successfully place tiles, using a specified position and rotation, using the tiles within a player's inventory, on their turn. After, using the tiles and placing the word, they will then be removed from the player's inventory, if there are tiles remaining in the `availableTiles` variable (bag), the `Dealer` procedure will fill the inventory with random tiles from the bag, to a sum of seven. However, if there are not any tiles available, the `Dealer` will not attempt to distribute any more tiles. In future developments the game will continue to play until a user has no tiles remaining, in which the loop condition will switch, and the game will end.

From testing the program so far, I have gathered a few issues within the code, including:

1. Attempting to place on the fifteenth row is regarded as an "Invalid coordinate". I thought this was the origin of a much larger issue. However, it was a simple programming error. Where, I had instead set the maximum value for validation as "14", when it should obviously be fifteen, after making this change, the fifteenth row is now accessible.

```
It's a's turn
['V', 'U', 'L', 'E', 'R', 'R', 'G']
Enter the word to placeruler
Enter: 1 for right, anything else for downwardl
Enter a coordinate e.g A5a15
Invalid coordinate
Enter a coordinate e.g A5a14
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 R U L E R + + + + + + + + + +
15 + + + + + + + + + + + + + + +
['V', 'G']
There are  80  letters left
There are now 75  letters left
Tap enter
```

```
#Checks if a passed input is in the correct formatting e.g B9
#of correct type and is within the specified range for both axis
def ValidateCoord(coord_in):
    validCoordsX = ["A","B","C","D","E"...]
    #If input is two or three digits
    if (len(coord_in) <= 3 and len(coord_in) >= 2):
        #Get substrings
        firstChar = coord_in[0].upper() #Convert to uppercase
        secondChar = coord_in[1:]

        #If first character is within valid coordinates
        if (firstChar in validCoordsX):
            #Check if second value is an integer
            if (CheckIfInt(secondChar)):
                #If within range 1-15
                if (int(secondChar) >= 1 and int(secondChar) <= 15):
                    return True
    return False
```

Adjustment to hardcoded value.

2. The second issue is more of a critical problem within the program, as it causes the program to break. It can logically be avoided, if a user considers each placement they make when nearing the board edges. For example, if a user enters a word that exits the board, an out-of-range error will occur, in other words, if a user were to enter "HI" at position O1 (top-right), and the word is orientated to the right, the program to attempt to place the character "I", at an index position within the 2D list that does not exist (outside of the grid).

```
    gameGrid[startPos[0]][startPos[1]] = x
IndexError: list index out of range
```

Error outputted, if this placement occurs.

The origin of this issue is within the `PlaceWord` method, as it does not regard the position that it is attempting to place the current letter. It places at index position: `[startPos[0],startPos[1]]`, ignoring what the values of those position actually are. Instead, of this I could adjust the method, to instead validate whether these positional values are within the range of the board. Otherwise, the method would return False, and `ProcessTurn` would recall. Allowing another input to be entered.

The problem with this solution is that the `PlaceWord` method is directly modifying the game board. Therefore, if a placement begins valid, e.g. in the "HI" example, the character "H", will be placed at position O1, but the letter "I" will cause the method to return False, thus recalling `ProcessTurn`. Meaning, that the letter "H" will remain on the board, despite the placement failing, also the tiles placed would not be deducted from the player's inventory, as the `PlaceWord` function would return false before it could reach the `LetterDeductor` method call.

A further solution to this new problem, is to have a temporary copy of the `gameGrid` variable (containing the game board), which the word would be placed in first, then if the word can be successfully placed, the word is then assigned to the main `gameGrid` variable. As well as resolving this issue, it would also make word validation much simpler as I would need to perform checks on the word after it has been placed, to then review the surrounding words for collisions. Attempting to achieve this type of validation using the main game board variable, would most probably cause many conflicts and raise further issues within the program, instead I will begin developing this method, and update the Gantt chart accordingly.

3. An issue that I have already addressed includes the lack of validation on the word input. Firstly, instead of the program detecting if the string that has been entered is producible using the current player's collection of tiles, then proceeding. It will assume the input is producible, and then attempt to deduct tiles even though they may not exist within the inventory, causing a `ValueError`. Proceeding from this, if the string entered is constructible using the tiles that the player's inventory contains, it should then be checked against a large dictionary of valid English words.

Lastly, to assess the word against its surrounding words, checking for conflicts and invalid combinations. The word must be placed in the current game board. For instance, creating a method to place a word within the main game board, then to perform checks on the combinations that the word will make (every word must connect in Scrabble, except for the starting placement), and if a resultant word is formed, that is not within the dictionary. I would then produce a method to remove the word from the game board, allowing the player to attempt their turn again.

Alternatively, I could produce a system like the one described above. That makes a copy of the latest game state, and then writes the word-in-question to this new copy, which I can then perform validation checks on. Achieving this would mean that if a word fails to place, I can then discard the temporary copy of the game board, and try again (or pass), this would make word validation simpler to implement. The only slight issue with this, is that each time a user takes their turn; a new instance of the game board will be created, meaning that a memory allocation is made to the stack and once the turn ends a de-allocation is made. This could be an issue if I was continuing to expand the size of the game board as the game progressed, or if a greater amount of data is stored within the two-dimensional list, causing larger allocation/de-allocation calls to be made. In reality, storing a list of this size has a negligible impact on memory usage, but it should always be kept in mind. In my case, a constant amount of data is being stored within the list. Firstly, being initialised with characters, and as the game progresses these characters are only overwritten, so memory usage is minimal.

4. As a short note, I forgot to include skip character (or space) detection within the `PlaceWord` method. I have implemented it within the `LetterDeductor` system, detecting if the current letter to be removed is a space, if so it is ignored. But, the same needs to be done for `PlaceWord`, in which a cell of the grid is skipped when a space character is met, then continuing. Instead of this, the program currently places the space character as if it is an alphabetic character, as you can see in the sample below, yet, successfully ignores the removal of the space character within `LetterDeductor`.

```
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + C L   N E + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

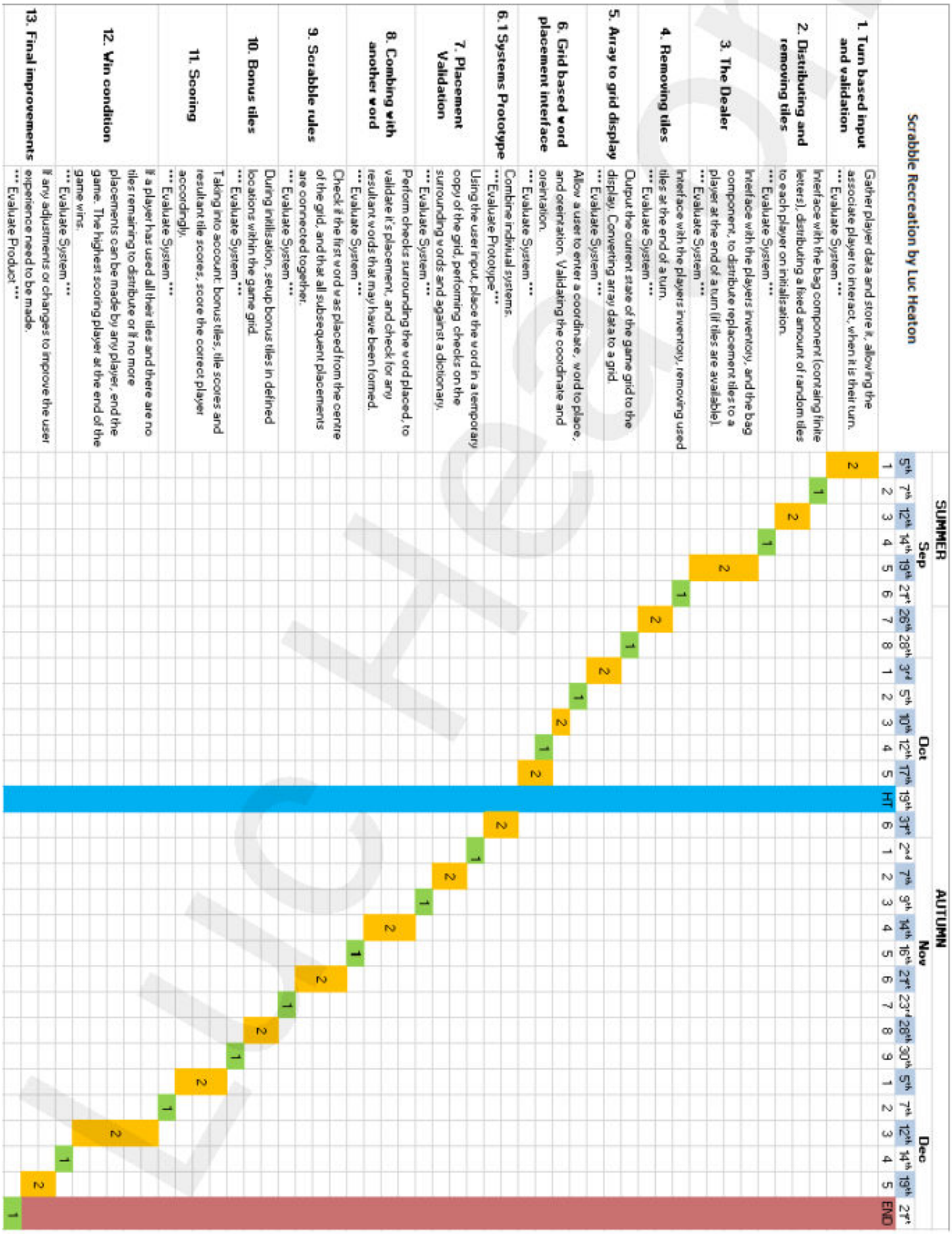
5. My last observations, are firstly the lack of a win condition. With the program continuing the main loop after a player has used all of their tiles. This could be a simple conditional placed after the call to the `Dealer`, confirming that there are no tiles available to distribute, checking if the player has any tiles remaining. If this is true, the game loop condition would then become false and the scores would be calculated. Here is a sample, showing the issue.

```
It's A's turn
[' ']
Enter the word to place
Enter: 1 for right, anything else for downward
Enter a coordinate e.g A5K9
   A B C D E F G H I J K L M N O
 1 G E O   V   L   I           +
 2 O   R E T B Z E O K I O R S +
 3 E D N U O I I T N Y P S E R +
 4 I T F L T E E O   C F O I M +
 5 U H G R W R I P V E M Y Q I +
 6 S I N R I C B N U I L D W T +
 7 E D E E N O G H I I S D N + +
 8   + + + + + + +   + + + + + +
 9 U + + + + + + + + +   + + + +
10 E + + + + + + + + + + + + + +
11 L + + + + +   X + + + + + + +
12 + + + + + +   + + + + + + + +
13 + + + + + + J + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
[' ']
There are  0  letters left
No tiles left in bag to distribute
There are now 0  letters left
Tap enter
It's B's turn
[' ']
Enter the word to place
```

5.1. The second observation being that, I am yet to include the usage of blanks within word input. Blanks currently have no usage, and so in future iterations I must detect if a player has a blank tile within their inventory and if they attempt to use a letter that they do not possess. Remove the blank tile from their inventory.

The win condition is theoretically met, yet the program continues to loop.

After, regarding the current state of the program, it is clear that the most prevalent issue is the lack of validation for word placements, including the lack of dictionary checking and observation of surrounding tiles. So, instead of continuing to the seventh system ("Combing with another word"), I will instead begin developing the validation systems. But, firstly I must update the Gantt chart to account for these changes, as the original does not include many of the smaller developments that have been highlighted within my evaluation, even overlooking the prototype, which was a necessary step in the development process.



With the chart updated, incorporating the latest full prototype, as well as some changes to time allocations. It is now time to begin designing and developing the placement validation system.

Sub system – Placement validation

Building upon my initial ideas, highlighted within my evaluation of the prototype, such as: creating a temporary copy of the current game board, which would be used to firstly place the word in. Acting as a safe test area, where validation checks can be made on the placement, without modifying the true game board. Then, depending on the outcome of the checks, the word can then be placed within the game board, removing the tiles used and scoring the player.

In addition to this, I must design and develop the "checks" that must be executed, firstly analysing the placement to find the full word that is formed due to the possible combinations of tiles (in the direction of the placement), then checking that "full word" against an English dictionary. As well as this, I must also develop the method(s) of checking the surrounding tiles around the placement, testing any additional words that may have been formed against the dictionary. However, to start with I will begin developing the method of creating a copy of the current game state, which can then be modified without modifying the original.

Development – Copying the board

My first thoughts, excluding the fundamental copying of the game board. Are the interactions that will be made with the created copy. For example, to actually modify the data within the grid, I would have to develop a new method of placing tiles within the grid, but after further thought, I can just adapt my existing `PlaceWord` method, adding another parameter which determines which grid the word is being written to, in a similar way to my `DrawGrid` method. This reuse of existing code within my program will save time and reduce the number of lines within the final program.

Original code

```
def PlaceWord(orientation,word_in,startPos):

    #Insert validation methods here
    #Temp capitalise
    word_in=word_in.upper()
    #If placed to the right
    if (not orientation):
        #For each letter in the inputted word
        for x in word_in:
            #Assign current letter to element within gameGrid
            gameGrid[startPos[0]][startPos[1]] = x
            #Increment position by one
            startPos[1]=startPos[1]+1
    #Opposite for downward
    else:
        for x in word_in:
            gameGrid[startPos[0]][startPos[1]] = x
            startPos[0]=startPos[0]+1
    #Update grid
    DrawGrid(gameGrid)
```

Adapted code

```
def PlaceWord(orientation,word_in,startPos,grid_in):
    #Insert validation methods here
    #Temp capitalise
    word_in=word_in.upper()
    #If placed to the right
    if (not orientation):
        #For each letter in the inputted word
        for x in word_in:
            #Assign current letter to element within gameGrid
            grid_in[startPos[0]][startPos[1]] = x
            #Increment position by one
            startPos[1]=startPos[1]+1
    #Opposite for downward
    else:
        for x in word_in:
            grid_in[startPos[0]][startPos[1]] = x
            startPos[0]=startPos[0]+1
    #Draw updated grid
    DrawGrid(grid_in)
```

Sub system breakdown and improvements

As you can see the adjustments are very minimal, also accounting for the changes throughout the code where the method is called. Yet, this now allows me to write to any grid, using the same method. Furthermore, as mentioned within my evaluation of the prototype within points two and four, I have forgotten to include the skip condition when a space character is detected while placing, causing empty spaces within the game grid when used. I can include this within the `PlaceWord` method as an additional condition before placing a tile, which should resolve the issue. Building upon this, I could also resolve the game breaking issue, when a placement attempt exits the board, however as highlighted within my method dependency considerations, I will develop this solution as one of the checks that is made within my placement validation, instead of performing the check within the `PlaceWord` method.

Adjusted code

```
def PlaceWord(orientation,word_in,startPos,grid_in):
    #Insert validation methods here
    #Temp capitalise
    word_in=word_in.upper()
    #If placed to the right
    if (not orientation):
        #For each letter in the inputted word
        for x in word_in:
            if (x == " "): #When a space is read skip a cell
                startPos[1]=startPos[1]+1
            else:
                #Assign current letter to element within gameGrid
                grid_in[startPos[0]][startPos[1]] = x
                #Increment position by one
                startPos[1]=startPos[1]+1
    #Opposite for downward
    else:
        for x in word_in:
            if (x == " "):
                startPos[0]=startPos[0]+1
            else:
                grid_in[startPos[0]][startPos[1]] = x
                startPos[0]=startPos[0]+1
    #Draw updated grid
    DrawGrid(grid_in)
```

A simple conditional, determining if the current letter within the input is equal to a space character. If it is, increment the current position by one. Otherwise, continue placing the letter as usual.

This was included, to account for both possible orientations, incrementing the position on the correct axis.

Original output

```
Enter the word to placeN LE
Enter: 1 for right, anything else for downward1
Enter a coordinate e.g A5g2
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + N   L E + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

New output

```
Enter the word to placeL D
Enter: 1 for right, anything else for downward1
Enter a coordinate e.g A5g2
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + L + D + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

Now, the space character will truly be used as a skip character. Rather than the program treating it as a character to place. Moving on from this, the development will now move onto creating a copy of the current state of the board. In other languages this would be a simple case of creating a new variable and assigning it the current value e.g. `copiedValue = originalValue` , however, when storing a variable within another, instead of creating a "copy" of the original value, assigning it to the `copiedValue`, it instead assigns a reference to the `originalValue` to the `copiedValue`. Meaning, that if a change is made to the `originalValue`, it will also be made to the `copiedValue`, e.g.:

```
Original grid
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + H + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

```
gameGrid=[["+"]*15 for i in range(15)]
gameGrid[5][5] = "H"
```

```
Copied and modified grid
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + H + + + + + + + + +
 7 + + + + + I + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

```
copiedGrid = gameGrid
copiedGrid[5][6] = "I"
```

```
Original grid
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + H + + + + + + + + +
 7 + + + + + I + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

```
print("Original grid")
DrawGrid(gameGrid)
```

As shown, despite the change only being made to `copiedGrid` it also updates the original grid.

Fortunatly, it is fairly simple to create a copy of most data types within python. However, in this specific case, where I need to make a copy of a two-dimentional list the simplest solution I have found, is to use a method from an external library called `copy` with the `deepcopy()` method. Unfortunatly, this method is fairly slow for computing standards, however to users it is barely noticable and it is a requirement for this method to work.

Though it is unnecissary to encapsulate this function call within another function, I will anyway, so that I can more easily track its usage throughout the program.

```
import copy
#Returns a copy of the grid passed
def CopyGrid(grid_in):
    return copy.deepcopy(grid_in)
```

Now, using this method instead of assigning the original value to the "copy". It can be seen below in this sample, that the copying works as expected:

```
Original grid
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + H + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

```
gameGrid=[["+"]*15 for i in range(15)]
gameGrid[5][5] = "H"
```

```
Copied and modified grid
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + H + + + + + + + + +
 7 + + + + + I + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

```
copiedGrid = CopyGrid(gameGrid)
copiedGrid[5][6] = "I"
```

```
Original grid
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + H + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

```
print("Original grid")
DrawGrid(gameGrid)
```

Development – Validation

With the method of creating a copy of the current game board working as intended, I can begin designing the method(s) of validation that will be executed, determining if a placement is truly valid or not. To start with, I looked into the general rules in which a word is deemed valid within the official game of Scrabble, which simplified states (paraphrased) that:

-The first player must form a word of length greater than two tiles, at a position that passes through the centre cell.

-Every subsequent player must form a word that uses an existing tile that is available on the grid, meaning that every word is connected by at-least one tile.

Example

In this example, the word "Horn" is connected to the word "Farms", by using the shared letter "r".

-If a tile is alongside any-other tile it must form a valid word.

Example

In this example, the new placement the word "Bit" is valid, because the resultant words created: "Pi" and "At" are also valid English words.

These rules are fairly straightforward and simple to follow when playing the physical board game, as it is quite easy to pick-up and move tiles, reading the result(s) in-front of you, and determine if the placement is valid. However, solving this same problem computationally, may-be more challenging. To overcome this, I have split the issue of validation into two groups: word-validation and surrounding-word-validation, this does not account for the dictionary check that must be made on the resultant word(s) formed.

Word validation, being the check of the full word, which would be the sum of the existing tiles that may connect to word, before, within or after, the inputted word.

Surrounding word validation, would be the validation that occurs after the full word is determined as being valid. Gathering the potential words that may be formed by adjacent tiles to the full word. Then the potential words and the full word, would then be checked against a dictionary. Finally, if all words formed are valid (the player would be scored for each word here,) the word is then placed in the original game board and the game continues.

Development – Word validation

Firstly, when the placement is made within the temporary copy of the grid, the word placed or "main word" should be checked, this would involve reading tiles in the direction of the placement, such as tiles that may appear before, and connect to the first letter of the inputted word, as it is possible to have existing tiles in-front of it. An example of this can be seen in this diagram I have produced below.

The letters in bold are the word to be placed, using the existing tile "F", before the placement.

Next, due to the nature that I have implemented the usage of an existing tile on the board (and the nature of the game), it is obviously possible that a placement can contain existing tiles within it (which within this project is achieved by using the space character in place of the existing tiles). Meaning, that I must cycle from the starting position of the word entered to the end of the input, reading from the temporary copy of the grid. Using this I can then construct a string, equal to the centre portion of the full word. An example of this can be seen in this diagram I have produced below.

The letters in bold are the word to be placed, using the existing tiles: "R" and "S", within the placement.

Lastly, it would be obvious to; as well as search before the placement, to also search after the placement. This would involve taking the last position of the placement and searching onward, in the direction of the placement- until a blank is hit. Using the sum of all three checks, the full word can be found and checked against a dictionary. An example of searching post placement can be seen in this diagram that I have produced below.

The letters in bold are the word to be placed, using the existing tile, "R", after the placement.

To begin with I am going to develop individual functions, performing each directional check. Each should return the tiles it has found, for example the function searching before the word (if tiles are present) should return the tile found, in-order of the direction e.g. H E **L L O** (bold letters are the placement), the before function should return "HE", if all the functions output in this format I could then concatenate the strings, forming the full word.

Development – Before word validation
As stated above I am going to develop an individual function for the three possible checks, which, when given the required arguments to locate and search the word placed, can return the connected, existing tiles found, or if not, the return would be an empty string. Having this collection of functions means that the "full word", including the players input as well as any existing tiles that may connect to the word (in the direction of the placement) can be determined.

My initial idea was to develop a small method that could be used to simplify the code for these validation checks. Instead of producing three complex functions that each have to evaluate the whole game board. I could alternatively produce a system that takes the slice (a row or column) of the game board that the word placement is within. This would mean that I could simplify the validation methods to only navigate and analyse a single-dimensional array (the slice), instead of a two-dimensional array (the game grid). Here is a diagram to further illustrate this system.

Game Grid

So, for this small game grid the letters in bold are the word to be placed. Passing the required arguments to the slicing method would return: `["M", "A", "N"]`, as it is the row that would be validated. As a further example, if the placement was only the letter "M" in the vertical direction, the function would instead return: `[" ","M", " "]`.

I feel as though this is a necessary development, that will improve the readability of my code, meaning future issues may be easier to identify and resolve, so I will suspend the development of the validation methods until this method is produced.

Sub system – Grid slicing
Again, as stated above this system will return the section or slice in the direction of the word placement. For this system to be usable by all three systems I would encapsulate the system, taking the game grid to be sliced as one parameter, and the starting coordinate and orientation inputs as another. Using these, the system would cycle through the row or column, in the direction of the placement, using the starting position to determine the row/column to be sliced. As the method cycles through the grid, each value in the section would then be stored within an array, which would then be returned as the slice.

Development – Grid slicing
The first step would involve locating the row or column that is being sliced. I could achieve this by using a combination of the orientation and the index-based, start position. Firstly, by determining if the orientation is right or downward, I can then select the appropriate axis within the start position (as the start position is a two-element array for the horizontal and vertical axis - in that order). For example, for a downward placement, I know that I require the horizontal index position (the first element within the start position), and the opposite is required for a rightward placement.

The reason that the index required, is the opposite to what is expected, is that, with the example of a downward placement, I need to know which column (the horizontal element) that the placement is within, then starting from the first row (the vertical element of index, zero) cycle iteratively until the end of the game board, which by default is fifteen iterations. So, with this I will now produce the function that will return the appropriate section, that will be used to perform the initial validation(s) on the word.

Code

```
#Given a grid to slice, the orientation of the slice and starting position
#Return the specified slice of the grid
def GrabSection(gridToRead, orientation_in, startPos_in):
    print(orientation_in)
    print(startPos_in)
    tempSection=[] #Stores section as it is being read
    #If right grab row
    if (orientation_in):
        for x in range(15):
            tempSection.append(gridToRead[x][startPos_in[1]])
    #If down grab column
    else:
        for x in range(15):
            tempSection.append(gridToRead[startPos_in[0]][x])
    return tempSection
```

Some temporary debug messages, displaying the given parameters that are used within the function.

I am using a hardcoded value, instead of a variable as I do not intend to alter the size of the game board, keeping the project as close to a true Scrabble recreation as possible.

Depending on the value of the orientation argument. The function will use the appropriate element (axis) from the index-based coordinate, and the iterating value of *x*, as the two index values required to navigate and access the game grid, in the format: `gridToRead[vertical component][horizontal component]`. The value contained within the grid at each position in the slice is then added to the end of the `tempSection` list, which is the returned as the sliced, section of the game grid.

Sub system evaluation

Despite the function working logically, I am still yet to test it for the intended output. As an additional note, as stated within the Validation considerations section, I am not implementing individual validation methods within each sub-system, such as this one, because it is unnecessary within this project, because I know that when the systems are combined, the inputs to the program will be validated within a function before they are passed any further. Returning back to the original point, of testing this system, I will test the method with validated data, which can simply be achieved by incorporating it within the latest full prototype, as all the current validation methods are in-place.

I decided the best location to make a temporary test function call to the `GrabSection` function is within the `ProcessTurn` method, as it contains all the required user inputs for arguments, even the index-based coordinate, which is required to locate the section to slice.

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    ...
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(validCoord)
    #Write to grid
    PlaceWord(orientation,validWord,indexedCoord)
    #Temp test call to grab section
    print(GrabSection(gameGridInUse,orientation,indexedCoord))
    #Remove used tiles
    LetterDeductor(playerLetters_in,validWord)
    #Re-distribute tiles
    Dealer(playerLetters_in)
```

Output

```
It's Luc's turn
['M', 'Y', 'H', 'E', 'T', 'V', 'O']
Enter the word to placeThem
Enter: 1 for right, anything else for downward1
Enter a coordinate e.g A5a1
   A B C D E F G H I J K L M N O
 1 T H E M + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
True
[4, 0]
['T', 'H', 'E', 'M', '+', '+', '+', '+', '+', '+', '+', '+', '+', '+', '+']
```

Specifying the user inputs, required, each turn.

The placement shown in the correct position within the grid, and the expected row to grabbed by the new function.

Debug messages, indicating True for a right facing orientation. However, the value displayed for the starting position is not the expected output. Where for a coordinate of `A1`, the index-based coordinate should be `0, 0`, yet it is shown to be `4, 0` and the section outputted is correct.

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    ...
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(validCoord)
    #Write to grid
    PlaceWord(orientation,validWord,indexedCoord)
    #Temp test call to grab section
    print(GrabSection(gameGridInUse,orientation,indexedCoord))
    #Remove used tiles
    LetterDeductor(playerLetters_in,validWord)
    #Re-distribute tiles
    Dealer(playerLetters_in)
```

```
def PlaceWord(orientation, word_in, startPos, grid_in):
    #Temp capitalise
    word_in=word_in.upper()
    #If placed to the right
    if (not orientation):
        #For each letter in the inputted word
        for x in word_in:
            if (x == " "): #When a space is read skip a cell
                startPos[1]=startPos[1]+1
            else:
                #Assign current letter to element within grid passed
                grid_in[startPos[0]][startPos[1]] = x
                #Increment position by one
                startPos[1]=startPos[1]+1
    #Opposite for downward
    else:
        for x in word_in:
            if (x == " "): #When a space is read skip a cell
                startPos[0]=startPos[0]+1
            else:
                grid_in[startPos[0]][startPos[1]] = x
                startPos[0]=startPos[0]+1
    #Update grid
    DrawGrid(grid_in)
```

The iterators, causing the issue.

Output

```
Enter the word to placewere
Enter: 1 for right, anything else for downward1
Enter a coordinate e.g A5g8
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + W E R E + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
True
[10, 7]
```

Unexpected *10* for the horizontal component within the index-based coordinate.

Sub system breakdown

After further investigation, I realised that the unexpected coordinate was caused by the same feature of Python that I have described within my Sub-System breakdown for the `LetterDeductor` method. In that, arguments for methods within Python are passed by reference, and not by value. Meaning, that within the `ProcessTurn` method, instead of the `PlaceWord` procedure making a copy of the `indexedCoord` variable, it instead creates a reference to it, and modifies the reference, which in-turn, modifies the original `indexedCoord` variable.

Dry running through the code output above, it can be seen that the index-based coordinate for the vertical axis (the second element) is the expected value of: seven (eight minus one as it is zero based). Whereas, the horizontal axis (first element) is larger than expected, when it should only be six (seven minus one). Backtracking through the `ProcessTurn` method, looking for the usage of `indexedCoord`, it can be seen that it is used within `PlaceWord`, where the variable `indexedCoord` (or local variable `startPos`) is used as an iterator for the length of the word, which explains why the unexpected output for the `indexedCoord` is equal to its expected value plus the length of the word i.e. 6 + 4 = 10.

Currently, the section output works as expected, outputting the full slice of the game grid with the word positioned relative to its position within the full grid, also containing the empty grid cells. However, despite the issue within the `indexedCoord` variable, where the alternate axis component, that is not required to output the correct slice, is being modified by the `PlaceWord` method. In other words, this doesn't cause any issues, but to prevent issues that may occur in future, I am going to resolve this issue, simply by making a "true copy" of the single dimensional `indexedCoord` variable, making use of the `copy` function attribute, which I have already imported into the project, from the copy library, during the development of Copying the board.

Adjusted code

```
def PlaceWord(orientation,word_in,startPos, grid_in):
    startPosTemp = startPos.copy()
    #Temp capitalise
    word_in=word_in.upper()
    #If placed to the right
    if (not orientation):
        #For each letter in the inputted word
        for x in word_in:
            if (x == " "): #When a space is read skip a cell
                startPosTemp[1]=startPosTemp[1]+1
            else:
                #Assign current letter to element within grid passed
                grid_in[startPosTemp[0]][startPosTemp[1]] = x
                #Increment position by one
                startPosTemp[1]=startPosTemp[1]+1
    #Opposite for downward
    else:
        for x in word_in:
            if (x == " "): #When a space is read skip a cell
                startPosTemp[0]=startPosTemp[0]+1
            else:
                grid_in[startPosTemp[0]][startPosTemp[1]] = x
                startPosTemp[0]=startPosTemp[0]+1
    #Update grid
    DrawGrid(grid_in)
```

Using the copy function attribute to create a "true copy" of the passed `indexedCoord` argument, which is then used instead of `startPos`.

Changes.

New output

```
Enter the word to placeYou
Enter: 1 for right, anything else for downward1
Enter a coordinate e.g A5a1
   A B C D E F G H I J K L M N O
 1 Y O U + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
True
[0, 0]
['Y', 'O', 'U', '+', '+', '+', '+', '+', '+', '+', '+', '+', '+', '+', '+']
```

The row expected to be sliced.

Now working, with the expected output of the *indexedCoord* variable. Displaying *0, 0* for the coordinate: *A1*.

The correct expected output of *GrabSection*, being a single dimensional array containing the word placed and the rest of the row.

Sub system breakdown and improvements

After developing the slicing function, I realised that I haven't determined the order of execution for the validation methods, including the fundamental development of basic validation of non-alphabetic characters for the user input. As well as, the incorporation of the duplication of the game grid, which the word is placed in first, for validation. Before, it is determined to be a truly valid placement. As an extra note, if any of these fields are not valid, an appropriate error should be displayed, and the player should be allowed to reattempt their turn, until a valid input is given, or if they pass. This is true for all the user input validation checks, such as coordinate input.

Development – Initial word input validation

So, to begin with, the user inputs need to be refined and finalised. Starting with the word validation. Currently, the word is simply accepted as the user input, when I must develop a method that only allows alphabetic characters, and spaces within the input for the word. In addition to this, the length of the input cannot logically be larger than fifteen, as it's the absolute maximum length of a word placement. The reason that the maximum is not seven, despite it being the maximum number of tiles available to a player at any time, is because it is theoretically possible to form a fifteen-letter word, by using the available tiles already placed on the board.

In short, these are the requirements and considerations for the word validation method:

-Must contain only alphabetic characters and spaces.
-Capitalisation should not have any effect on the outcome.
-The maximum user input is 15 characters.
-Allow the user to reattempt the input on fail.
-Allow the user to pass their turn.
-Return meaningful and useful errors.

Code

Usage of the escape line character: *\n* to make the output clearer.

Usage of the escape character: \ to allow double quotations within the string.

Convert to uppercase to ensure inputs are not affected by case.

```
def ValidateWordInput():
    while True:
        #Retrieve word from user in uppercase
        wordAttempt = input("Enter the word to place\nor \"pass\" to forfeit your turn ").upper()
        #First check for pass
        if (wordAttempt == "P" or wordAttempt == "PASS"):
            return False
        #If string contains only spaces and alphabetic characters
        if (all(letter.isalpha() or letter.isspace() for letter in wordAttempt)):
            if (len(wordAttempt) <= 15): #If less than 15 chars
                return wordAttempt #Return uppercase validword
            else:
                print("The word entered is greater than 15 characters")
        else:
            print("The word must ONLY contain letters and spaces")
```

The function returns False if the user enters wishes to forfeit their turn, which will be handled by the function call.

Appropriate error messages.

```
if (all(letter.isalpha() or letter.isspace() for letter in wordAttempt)):
```

I have separated this line of code as it's slightly more complex than the rest (being mostly conditionals). It works by iterating through the string (`wordAttempt`), character at a time, where the current letter being iterated is equal to `letter`. During each iteration, the value of `letter` is checked if it is either (using the `or` keyword) an alphabetic character or a space. If all of the iterations return `True` from either (or both) of those checks, by using the `all` function, `True` will be returned.

The same can be achieved using a different method like this one I've created; however, the shorthand is useful in this case.

```
def isValid(string):
    for letter in string:
        if (not (letter.isalpha() or letter.isspace())):
            return False
    return True

string = "Hello"
print(isValid(string))
```

=

```
print(all(letter.isalpha() or letter.isspace() for letter in string))
```

To allow the user to reattempt their input, I have simply placed the code block within an infinite loop, which may be seen as a poor programming practice, however, it is a suitable solution in this case, due to the usage of function return(s) to escape the loop, either when the user wishes to forfeit their turn or when a valid input is given.

I also came to the realisation that it is obviously possible (and valid) for a user to request to place the word "pass" or even the letter "p". So, to resolve this issue entirely, I am going to require that the input contains a character that cannot be found within a valid word placement, such as a symbol, or in this case, an exclamation mark.

Adjusted condition

```
if (wordAttempt == "PASS!"):
```

Also, updating the print statement accordingly.

With this method complete, I can now incorporate it within the `ProcessTurn` method, within the current build. Instead, of assuming that the user input is valid.

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    returnedWordInput = ValidateWordInput() #Gather word input or check for pass
    #If entered pass, move onto next player
    if (not returnedWordInput):
        return
    #Otherwise continue
    ...
```

Sub system breakdown

By simply, storing the return of the function call in a variable. I can then test within a conditional, whether or not the user has entered the pass statement, if so, then the procedure will `return` (or break), preventing any further inputs from being requested, and the next player takes their turn.

For the case of orientation input, I feel as though there is not much to change. As the input is not variable, it can only be one or the other. However, I think that it would be suitable to allow the user to reattempt their turn at this point, in the case that the user changes their mind. With the same feature added to coordinate input.

Sub system – Managing user inputs

To achieve this, I will have to redesign the `ProcessTurn` procedure to allow the input sections to be recalled at any point, without altering the player's inventory or the game board. The method that I have in mind, is to move all the input calls from `ProcessTurn` to a function that manages all three, as well as moving the inputs to their own function, which returns the user's inputs within an array, in a similar way to the `ConvertCoordToIndex` function, then by using returns, I can detect different conditions. For example, in the case of a user passing their turn during the word input stage, break the execution and move onto the next player, or in the case of a reattempt, within the orientation input stage, the function that all the inputs are contained within would be recalled.

To begin with, I will remove the function handling for each input type from `ProcessTurn` into another called `RetrieveInputs`, this will allow it to be recalled. Then, the various input managers for each input, such as

basic word validation will be moved to a separate function, that returns appropriate, detectable values when an event, such as a pass turn, in which the `ProcessTurn` procedure will cancel execution, allowing the next players turn to be made. Being more specific, I need to detect for multiple different events, including the event, in which a return of an array containing all the valid inputs is given. To achieve this, I could return hardcoded values, for example the number one could be returned, if the user requires a reattempt, or the number two, if the user wants to skip their turn. However, this method makes the readability of my solution much harder to understand.

Development – Managing user inputs using classes

Due to the fact, I have to account for multiple different events. I cannot rely on Boolean returns as they only have two states (excluding nullable), and I cannot use magic numbers as they are meaningless within the program. A solution to this is a class, where I can assign meaningful variable names to magic numbers, which represent each event. For example, within a class I could have a variable named `SKIPTURN`, which represents the constant value of one, instead of the function returning the number one, it would instead return `SKIPTURN`. Practically, I could just create a selection of variables for each event type, instead of encapsulating them within a class, however, grouping all the event types within the same class will make its usage throughout the program clearer.

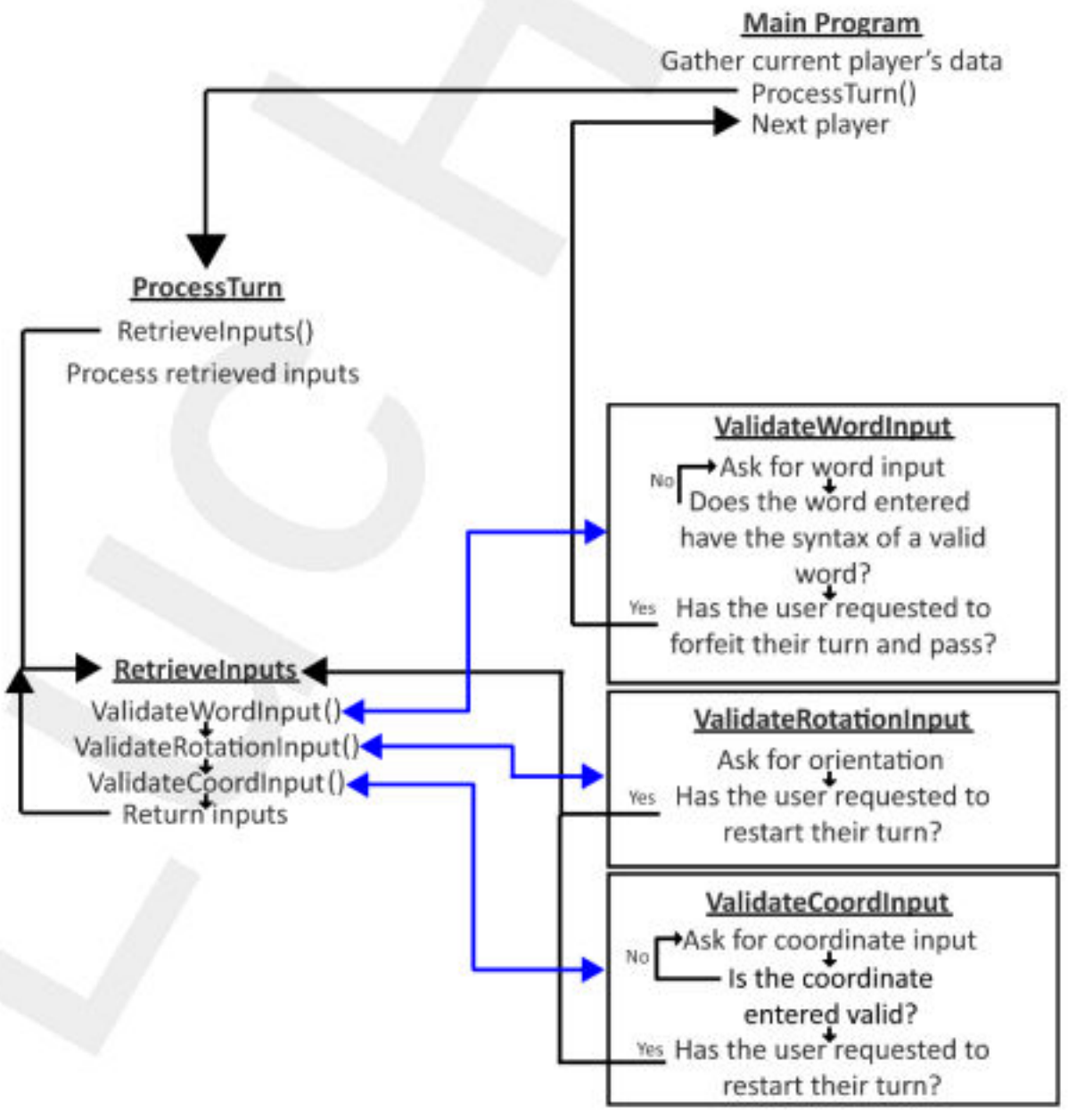
So, to begin with. It is clear that this class will only be a storage of variables and so it will not require a constructor. It will instead have a list of declared values with meaningful names. For this project, I will create a simple class called `Event`, which will include definitions for these events:

SKIPTURN – Break at this point within `ProcessTurn`, and continue to the next player.

RESETTURN – Break at this point within `ProcessTurn`, and recall input managers, allowing the user to reattempt their turn.

Turn procedure diagram

To better illustrate the method in which, I intend to implement these changes to the inputs systems, I have produced a diagram, showing the interactions of systems, starting from the main program.



As you can see from the diagram, some of the functions, such as `ValidateRotationInput`, do not, currently exist. To keep the program consistent, I must firstly, separate individual input checks into their own functions, that also include the condition for restarting or skipping a turn.

This must be completed for orientation input as well as coordinate input, as word input has already been compartmentalised within `ValidateWordInput`. For the case of coordinate input, the method for validation has already been developed within `ValidateCoord`, however, I also need to produce a `ValidateCoordInput` which will continuously ask the user for a valid input or a turn restart. Which will mostly consist of the code that is already present within the `RetrieveInputs` method, shown here:

```
def RetrieveInputs():
    ...
    while True:
        validCoord = input("Enter a coordinate e.g A5")
        #If coordinate is valid
        if (ValidateCoord(validCoord)):
            break
        else:
            print("Invalid coordinate")
```

In other words, I will be moving this code sample from `RetrieveInputs` to its own function called `ValidateCoordInput`, which will also account for the reset of turn inputs. And the same for `ValidateRotationInput`.

Code

```
def ValidateWordInput():
    while True:
        #Retrieve word from user in uppercase
        wordAttempt = input("Enter the word to place\nor \"pass!\" ...").upper()
        #First check for pass
        if (wordAttempt == "PASS!"):
            return 1 #Temp return for skip turn, before class is developed
        #If string contains only spaces and alphabetic characters
        if (all(letter.isalpha() or letter.isspace() for letter in wordAttempt)):
            if (len(wordAttempt) <= 15): #If less than 15 chars
                return wordAttempt #Return uppercase validword
            else:
                print("The word entered is greater than 15 characters")
        else:
            print("The word must ONLY contain letters and spaces")

def ValidateRotationInput():
    while True:
        orientation = input("Enter \"1\" for right...").upper()
        if (orientation == "RE"): #Reset inputs
            #Temp return for reset turn, before class is developed
            return 2
        elif (orientation == "1"): #Rightward placement
            return True
        elif (orientation == "2"): #Downward placement
            return False
        else:
            print("Invalid input, please try again")
def ValidateCoordInput():
    while True:
        validCoord = input("Enter a coordinate ...").upper()
        if (validCoord == "RE"): #Reset inputs
            #Temp return for reset turn, before class is developed
            return 2
        #If coordinate is valid
        if (ValidateCoord(validCoord)):
            #Breaks loop and returns valid value
            return validCoord
        else:
            print("Invalid coordinate")

def RetrieveInputs():
    #Gather word input or check for pass
    returnedWordInput = ValidateWordInput()
    #If entered pass, stop processing for this player
    if (returnedWordInput == 1): #Temp detection before class is developed
        return 1
    #Otherwise continue
    #Gather rotation input and check for reset
    returnedRotInput = ValidateRotationInput()
    if (returnedRotInput == 2): #Temp detection before class is developed
        return 2
    #Otherwise continue
    #Gather coordinate input and check for reset
    returnedCoordInput = ValidateCoordInput()
    if (returnedCoordInput == 2): #Temp detection before class is developed
        return 2
    #Otherwise continue
    return [returnedWordInput,returnedRotInput,returnedCoordInput]
```

Code breakdown

Firstly, I've included the conditional to the `ValidateWordInput` method, using the magic number 1 as a placeholder for the `SKIPTURN` event. It can be noted how meaningless the number currently is, until I replace it with the `Event` class.

Secondly, I compartmentalised both methods for retrieving and validating coordinate and orientation inputs. Which simply involved moving the code seen above from `RetrieveInputs` to its own individual function, with the addition of a substitute skip or reset event, represented by a 1, or a 2, for skip and reset, respectively.

Continuing, I also had to adapt the `RetrieveInputs` function, to account for the return of each individual input. As well as return the for a set of successful inputs, in which an array containing all three valid values are contained.

Lastly, changing the `ProcessTurn` method to account for this structured array output.

Usage of placeholder values, to be replaced with events.

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    #Gather user input
    userInputs = RetrieveInputs()
    #Handle skip turn and reset turn here

    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(userInputs[2])

    #Write to grid
    PlaceWord(userInputs[1],userInputs[0],indexedCoord,gameGridInUse)

    #Remove used tiles
    LetterDeductor(playerLetters_in,userInputs[0])
    #Re-distribute tiles
    Dealer(playerLetters_in)
```

Using indexes for structured data arrays is not ideal programming practice, as it requires a background knowledge or comment explaining it.

Sub system improvements

As an alternative to an array return, I could instead create a class with a constructor which would contain the attributes: *wordInput*, *rotationInput* and *coordInput*. This would be better programming practice than relying on the knowledge that specific indexes within the return are equal to specific inputs within the structured array.

Next, I will implement the *Event* class, replacing the placeholder conditionals and returns within the input functions above, as well as a class to replace the array structure, to improve readability.

Code

```
#Stores "constant" reference values for events
class Event:
    SKIPTURN = 1
    RESETTURN = 2

def ValidateWordInput():
    ...
        if (wordAttempt == "PASS!"):
            return Event.SKIPTURN
    ...
def ValidateRotationInput():
    ...
        if (orientation == "RE"): #Reset inputs
            return Event.RESETTURN
    ...
def ValidateCoordInput():
    ...
        if (validCoord == "RE"): #Reset inputs
            return Event.RESETTURN
    ...
#Stores data about the user inputs
class InputData:
    #Constructor, assign values of each attribute
    def __init__(self,word,rotation,coord):
        self.word = word
        self.rotation = rotation
        self.coord = coord
def RetrieveInputs():
    #Gather word input or check for pass
    returnedWordInput = ValidateWordInput()
    #If entered pass, stop processing for this player
    if (returnedWordInput == Event.SKIPTURN):
        return Event.SKIPTURN
    #Otherwise continue
    #Gather rotation input and check for reset
    returnedRotInput = ValidateRotationInput()
    if (returnedRotInput == Event.RESETTURN):
        return Event.RESETTURN
    #Otherwise continue
    #Gather coordinate input and check for reset
    returnedCoordInput = ValidateCoordInput()
    if (returnedCoordInput == Event.RESETTURN):
        return Event.RESETTURN
    #Otherwise continue
    #Create and return new instance of input data
    return InputData(returnedWordInput,returnedRotInput,returnedCoordInput)
```

Code breakdown

Due to the fact Python does not have a built-in read-only access modifier (excluding external libraries), I will instead mark the variable as a constant, using the standard uppercase formatting.

With the implementation of the *Event* class, I've modified each placeholder value with the appropriate class attribute. Which is notably understandable when compared to magic numbers.

This is the basic class that I have developed to store the returned input data, as a replacement for the structured array. It contains three attributes: *word*, *rotation* and *coord*, which can be referenced instead of array indexes, for example, *InputData.word* can be used, instead of *RetrieveInputs()[0]*, as you can see the former is much clearer.

Also, replacing the placeholder conditions and returns, with the class attributes.

Replacing, the old return of a structured array, with the syntax to instantiate a new instance of the *InputData* class.

It can be seen that the *InputData* class has a method called *__init__*, this is the constructor. It is a function that can contain any sort of task to be executed on its creation. It is automatically called when an instance (a unique version) of the class is created using the syntax *InputData(value1,value2...)*. It contains an argument named *self*, which contains a reference to this specific instance of the class, and is required by the constructor, without it, it would not be possible to have multiple instances of the class. Within the constructor, I have assigned the given parameters to the associate attributes. Meaning, that when I reference the new return of the *RetrieveInputs* function, I can now access its attributes using the equivalent of *RetrieveInputs.word* for each attribute.

With all of these changes to improve the programs readability complete, the final step for implementing these user-friendly mechanisms can be made. Following the Turn procedure diagram that I produced earlier, it can be seen that the final step, is the processing of user inputs within `ProcessTurn`, as all of the returns and conditions for `SKIPTURN` and `RESETTURN` are now in place.

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    #Gather user input
    userInputs = RetrieveInputs()
    #Handle skip turn and reset turn here
    ...
```

This involves creating conditionals within `ProcessTurn` handling the possible returns. The possible returns include: `Event.SKIPTURN`, `Event.RESETTURN` or an instance of the newly developed `InputData` class.

To begin with, handling `Event.SKIPTURN` is the simplest, as I can simply break out of the `ProcessTurn` procedure, which will prevent further processing for this player, and will move onto the next player. However, `Event.RESETTURN` is slightly more convoluted to handle. I cannot simply recall `RetrieveInputs`, because it would not allow the player to reattempt their turn more than once, and there will be repeated code.

The simplest solution to this, is one that I have used many times throughout the program. Loops. Introducing an "infinite" loop, will allow the program to request an input and handle returns more than once- if required. For example, the execution will remain in the loop, if a user requests to reattempt their turn. Meaning, that the `RetrieveInputs` function would be recalled and all the returns will be handled again. In the case of a user wanting to forfeit their turn, the execution of `ProcessTurn` (and loop) would halt, using a `return` statement. Finally, for a successful input, the `break` statement could be used to exit the "infinite" loop and continue the processing of the user's inputs.

Code

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    while True:
        #Gather user input
        userInputs = RetrieveInputs()
        #Handling returns
        if (userInputs == Event.SKIPTURN):
            print("Forfeiting turn")
            return
        elif (userInputs == Event.RESETTURN):
            print("Reattempting inputs")
        elif (type(userInputs) == InputData):
            break
    #Continue processing
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(userInputs.coord)
    #Write to grid
    PlaceWord(userInputs.rotation,userInputs.word,indexedCoord,gameGridInUse)
    #Remove used tiles
    LetterDeductor(playerLetters_in,userInputs.word)
    #Re-distribute tiles
    Dealer(playerLetters_in)
```

Code breakdown

It can be seen that each return type is being handled, with the exception of a user reattempting their turn, where only an output is given. This is because the execution must stay within the loop, to allow the `RetrieveInputs` function to be recalled.

The condition for handling a successful return of the class is slightly more complex, as I have to verify if, it is an instance of the `InputData` class, and nothing else. After looking online, I learnt that the built-in, `type` function, which I understood to return the data type of the object given such as identifying and number as an integer, also works with classes, returning a unique type for each class. For example, `type(ClassA)` returns a different value to, `type(ClassB)`. So, for this use case. I am simply comparing if the type returned from `RetrieveInputs`, is equal to the `InputData` class. If so, then by using the `break` statement, the program can exit the "infinite" loop and process the given inputs.

Lastly, the usage of the attributes from the `InputData` class within the processing methods, makes the meaning of the code much clearer, in comparison to the old method which took the form of: `userInputs[x]`, which has no meaning, without explanation or background understanding.

Output – Skip turn

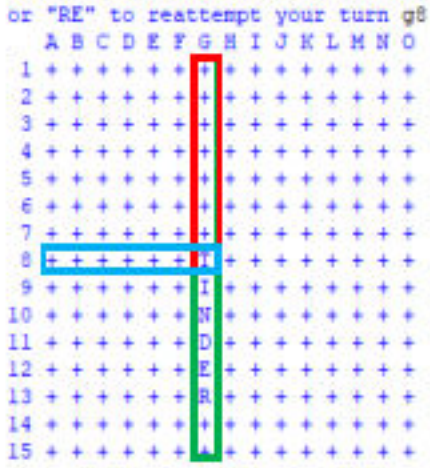
```
Enter the number of players 2
Enter player name Luc
Enter player name Will
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
It's Luc's turn
['I', 'D', 'L', 'P', 'X', 'N', 'U']
Enter the word to place
or enter "pass!" to forfeit your turn pass!
Forfeiting turn
Tap enter
It's Will's turn
['N', 'E', 'I', 'S', 'T', 'Y', 'E']
Enter the word to place
or enter "pass!" to forfeit your turn
```

Code breakdown

If a user enters the "pass!" statement, the `ValidateWordInput` function will return `Event.SKIPTURN`, which is then passed down the function chain, outputting "Forfeiting turn" and breaking out of the `ProcessTurn` function.

Output – Reset turn

```
It's Luc's turn
['E', 'R', 'T', 'T', 'I', 'D', 'N']
Enter the word to place
or enter "pass!" to forfeit your turn TIDE
Enter "1" for right, "2" for down
or "RE" to reattempt your turn re
Reattempting inputs
Enter the word to place
or enter "pass!" to forfeit your turn tinder
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn g8
```



- Grid slice for this placement
- Horizontal component: 8
- Vertical component: G

At the rotation input stage, the user can input "re" (capitalisation has no effect), which causes `ValidateRotationInput` to return the `Event.RESETTURN` attribute, which, again is passed down the function chain, outputting a meaningful message. Then restarting the player's turn, allowing them to re-enter their desired input.

It can also be noted that the program still works as expected, placing the word in its specified location while using the `InputData` class attributes.

Development – Sandbox test environment

To tie up the final loose end before resuming development on the validation method, I must setup the test environment which is used to perform the validation checks safely on the placement (without effecting the game). This will be a case of using the `CopyGrid` function that I developed earlier within the project, as well as the adapted `PlaceWord` method. By creating a true duplicate of the current game board, the word can be placed within it, and this duplicate version of the `gameGrid` can then be used by the `GrabSection` function, to gather the required slice, that can be used in the validation methods that will soon be developed.

The first process will involve storing the duplicate of the main game board, then passing it, along with the required input data from the user, to the `PlaceWord` method. I can then grab the slice of this duplicate game board, which now contains the placed word, it will then be used by the validation methods. As these subroutines require the inputs from the user, it would make sense to insert these methods within `ProcessTurn`.

Code

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    ...
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(userInputs.coord)

    #Create copy of gamegrod
    sandboxGrid = CopyGrid(gameGridInUse)
    #Write word to sandbox grid
    PlaceWord(userInputs.rotation,userInputs.word,indexedCoord,sandboxGrid)
    #Grab the section
    placedSection = GrabSection(sandboxGrid,userInputs.rotation,indexedCoord)
    print(placedSection)
    ...
```

To reiterate the method from above. The main game grid variable is copied and assigned to `sandboxGrid`, which is then used by `PlaceWord` to make the safe placement of the word. With the word placed in its specified position, the section that will be used by the validation methods can be gathered by the `GrabSection` function.

Output

```
It's Luc's turn
['E', 'D', 'R', 'E', 'E', 'G', 'I']
Enter the word to place
or enter "pass!" to forfeit your turn red
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn f3
['+', '+', '+', '+', '+', 'R', 'E', 'D', '+', '+', '+', '+', '+', '+', '+']
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + R E D + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

It can be seen that the section (outlined in green) from the `sandboxGrid` variable contains the word to place, in its correct position, relative to the `gameGrid`. This can now be used to perform the validation checks for before, within and after the placement.

As an additional note, the location of the method call for `DrawGrid` has been moved to the main program game loop. Without this change, the grid would have been displayed multiple times whenever `PlaceWord` is called.

With all the recent, smaller developments required to simplify the development process as well as quality-of-life changes, I feel it is required to summarise all of the features that have been included:

Front end
-Users can skip/forfeit/pass their turn when entering the word to place
-Users can reattempt/reset their turn when entering either the rotation or start coordinate
Back end
-Introduction of the `Event` class to keep track of the usage of skip turn and reset turn, also improving code readability and maintainability.
-Introduction of the `InputData` class as a temporary storage for returned user input, using its attributes to make accessing specific components of the user's input, such as rotation easier to track.
-Each user input system has been compartmentalised and standardised, making the code easier to follow.
-The duplication of the game board, which can be used to test the placement before it is placed within the main game board.

Resumed Development – Before word validation
Now that I have developed the prerequisites required to make the development for the validation methods, simpler. I can now begin the development of the pre-search function, that is, to search before the word placement. The method that I am considering to produce, to achieve this is, to locate the position of the first letter within the section, containing the placement, and store all the tiles that may be found. Which would simply be accomplished by using the entered coordinate from the user, then iterating backwards, away from the placement, storing the contents of each cell, until an empty cell is hit. This would mean that the result is in reverse (due to the backward iteration), and so the final result would have to be reversed.

Firstly, using the index-based version of the user entered coordinate, and the slice of the `gameGrid` variable which the placement is contained. The program will take the required component of the index-based coordinate, which is dependent on the orientation of the word, for example, the index position within the slice, for the first character within a horizontal placement will be equal to the vertical component (or first element) of the index-based coordinate. This position will include the first character within the placement, so the backwards iteration must begin from the vertical component minus one (using the example above). It should loop until it either hits a blank space within the grid, which is illustrated by a "+" character, or if it reaches the zeroth index. Each character that is collided with can be stored within a list during iteration, which is finally reversed and returned.

Code

```
def BeforeWordTiles(section_in,indexcoord_in,rotation_in):
    #Determine index required
    indexPointer = 0
    if (rotation_in): #Right placement
        indexPointer = indexcoord_in[0]-1
    else: #Down placement
        indexPointer = indexcoord_in[1]-1
    #If equal to -1/placement at border
    #return empty
    if (indexPointer < 0):
        return ""
    charsBehind=[]
    #Loop if current cell is not empty and hasn't reached -1 index
    while (section_in[indexPointer] != "+" and indexPointer >= 0):
        #Store value of current index in list
        charsBehind.append(section_in[indexPointer])
        #Decrement index
        indexPointer-=1
    #Reverse the list and return as a string
    return ''.join(reversed(charsBehind))
```

Code breakdown

The first task, is to determine which component of the index-based coordinate array is required to locate the first character within the slice. Deducting one from this value to equal the index for the cell that may/may not exist before the first character. Then, to prevent an `IndexError` from occurring, when the start position of a placement is already at the zeroth index, on either the horizontal or vertical axis the `indexPointer` will equal -1 (due to the deduction of 1) and so an empty string will be returned.

If the execution gets past the conditional, that means, that it is possible for a/multiple tile(s) to exist before the first letter of the placement. The function will then loop conditionally if: the current index position is not equal to an empty cell (which contains a "+" character) AND the current index iterator is greater than or equal to zero, meaning that it is within the slice. If these conditions are true. Append the content of the slice at the index before the first letter in the placement, within the `charsBehind` list, then for each iteration deduct the index by one, and store the slice content.

When the loop finishes, in the case that there were tiles before the placement. The `charsBehind` list will contain the tiles found, in reverse order, due to the backward iteration. So finally, using a useful built-in function called `reversed`, the contents of the list given are returned in reverse order, giving the result in the correct order. Pairing this with `''.join`, allows a list to be converted to a concatenated string. It works by using the contents within the `''` as a separator between each element within the list given, and because I simply want the result of the elements concatenated as a string, for example, the list: `["H","E","L","L","O"]` as `"HELLO"`, I simply specify an empty string.

To see the function working, I simply added a statement that performs the before check on the slice of the sandbox variable, passing the required arguments.

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    ...
    #Grab the section
    placedSection = GrabSection(sandboxGrid,userInputs.rotation,indexedCoord)
    print(placedSection)
    print(BeforeWordTiles(placedSection,indexedCoord,userInputs.rotation))
    ...
```

Sub system breakdown

Here you can see the output of `BeforeWordTiles`, working perfectly, as expected. Shown firstly, by the empty print statement, which is returned because the index of the character, before the first letter within the placement is equal to -1, causing the function to return an empty string. When the user requests the placement of "d" at the coordinate "d1", the program grabs the correct slice of the sandbox grid, containing both; the existing word "bon" and the new placement of "d". Giving this slice to `BeforeWordTiles`, returns as expected "BON", which is equal to the tiles the exist directly before the placement.

Output – Horizontal placement

```
It's Luc's turn
['D', 'T', 'L', 'C', 'N', 'B', 'O']
Enter the word to place
or enter "pass!" to forfeit your turn bon
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn a1
['B', 'O', 'N', '+', '+', '+', '+', '+', '+', '+', '+', '+', '+', '+', '+']

   A B C D E F G H I J K L M N O
 1 B O N + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
Tap enter
It's Luc's turn
['D', 'T', 'L', 'C', 'E', ' ', 'E']
Enter the word to place
or enter "pass!" to forfeit your turn d
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn d1
['B', 'O', 'N', 'D', '+', '+', '+', '+', '+', '+', '+', '+', '+', '+', '+']
BON
   A B C D E F G H I J K L M N O
 1 B O N D + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
Tap enter
```

Output – Vertical placement

```
It's Will's turn
[' ', 'F', 'E', 'S', 'E', 'C', 'I']
Enter the word to place
or enter "pass!" to forfeit your turn s
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn g5
['+', 'T', 'I', 'E', 'S', '+', '+', '+', '+', '+', '+'...
TIE
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + I + + + + + + + +
 3 + + + + + + I + + + + + + + +
 4 + + + + + + E + + + + + + + +
 5 + + + + + + S + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

Output – Not connected placement

```
It's Rob's turn
['Y', 'D', 'E', 'X', 'I', 'O', 'U']
Enter the word to place
or enter "pass!" to forfeit your turn ox
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn 14
['+', '+', 'T', 'I', 'L', 'E', '+', '+', 'O', 'X', '+', '+', '+', '+', '+']

   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + T I L E + + O X + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

It can also be seen that the function works for a vertical placement as well, returning "TIE" when placing the letter "s". In the situation, where a placement shares the same row/column with an existing placement, it is shown that the function detects the appearance of a "+" character, that separates the two placements. Returning any characters that it has found before it has collided with the "+" character, which in this example is nothing.

Sub system – Within word validation

With the first validation method, fully developed and implemented it is now time to develop the second method. Requiring the validation of the possible existing tiles within a placement. In the case where a user enters, for example: "T E", where the second character is skip character, declaring that it will be filled with an existing tile on the board. In this example, there is an existing vertical placement, which this user is using the letter "H", to form the word "THE". As this is possible, a check must be made through the length of the placement, also reading from the game board.

Development – Within word validation

In a similar way to the Before validation method above, this function must iterate through the slice containing the placement. But, instead of performing any checks for empty tiles and reversing the outcome. This function simply starts from the start position of the user input and iterates through the length of the word, storing the element from each cell within a list, and returning that list. Doing this, means that in place of any skip characters, the existing tiles being used can be grabbed.

Code

```
def WithinWordTiles(section_in,word_in,indexcoord_in,rotation_in):
    #Determine index required
    if (rotation_in): #Right placement
        indexPointer = indexcoord_in[0]
    else: #Down placement
        indexPointer = indexcoord_in[1]
    fullWord=[]
    #Iterate for amount of chars within word
    for x in range(len(word_in)):
        #Store element at index position (iterator+start index)
        fullWord.append(section_in[x+indexPointer])
    return ''.join(fullWord)
```

Output

```
It's Luc's turn
['N', 'C', 'T', 'J', 'E', 'I', 'G']
Enter the word to place
or enter "pass!" to forfeit your turn G T
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn f4
['+', '+', '+', '+', '+', 'G', '+', 'T', '+'...

G+T
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + G + T + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

Code breakdown

Firstly, I had to determine the component of the index-based coordinate required to locate the position of the starting letter of the placement. Then simply using the length of the word, found using the `len` function, which in this case returns the number of characters within the string given. I have then used the sum of the `indexPointer`, used as a static variable for the starting index of the word, as well as the iterating value of `x`, to cycle through the slice. Appending the content of each index within the `fullWord` list. Which is then returned as a concatenated string.

Sub system breakdown

Analysing the output of the code, it may seem that the output, is not the output that the program would want. Due to the occurrence of an empty cell character "+". The existence of this character within the placement means that the placement must be invalid, as it counts as two separate placements. However, instead of handling this invalid placement within this function. The program could instead continue validation, then, before the program performs the dictionary check on the full concatenated version of the input (i.e. the result of the concatenation of before + within + after). It can firstly check for the occurrence of the empty cell character within the full string, in which the placement is determined as invalid.

Other than this consideration, the method works as expected. Successfully reading the characters from the slice of the sandbox game board.

Sub system – After word validation

The final validation method should also be fairly simple, as it essentially performs the opposite of the Before word validation method. Cycling from the index after the last character within the word placement, storing each cells content, until it hits a blank tile or the edge of the board. Returning, the result of the potential contents found, as a string.

Development – After word validation

As stated above, the design for this method is very similar to Before word validation. However, the index that the iteration begins from, is not the location of the first character. It would instead equal the starting index, plus the length of the string. There is no need to add one, because the sum of the length of the word, and the start index, is equal to the index of the next character.

Unlike the Within word validation, this method must contain an initial check in the case that the index is at the right/bottom edge of the board, in a similar way to Before word', to prevent the possibility of an `IndexError`, the same must be done to the iterating loop, preventing an inaccessible index from being accessed, which would throw the same error.

Code

```
def AfterWordTiles(section_in,word_in,indexcoord_in,rotation_in):
    #Determine index required
    if (rotation_in): #Right placement
        indexPointer = indexcoord_in[0]+len(word_in)
    else: #Down placement
        indexPointer = indexcoord_in[1]+len(word_in)
    #If equal to 15/placement at border
    #return empty
    if (indexPointer > 14):
        return ""
    charsAfter=[]
    #Loop if current cell is not empty and hasn't reached the border
    while (section_in[indexPointer] != "+" and indexPointer < 15):
        print(indexPointer)
        #Store value of current index in list
        charsAfter.append(section_in[indexPointer])
        #Increment index
        indexPointer+=1
    #Return as a string
    return ''.join(charsAfter)
```

Output

```
It's Will's turn
['F', 'O', 'N', 'I', 'C', 'E', 'E']
Enter the word to place
or enter "pass!" to forfeit your turn n
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn m1
['+' ... '+', 'N', 'O', 'W']
N
13
14
Traceback (most recent call last):
  File "E:\Py\Scrabble\Documentation\Sub Systems\_WordPlacingBuild
ing.py", line 415, in <module>
    ProcessTurn(playerInventories[currentTurn],gameGrid)
  File "E:\Py\Scrabble\Documentation\Sub Systems\_WordPlacingBuild
ing.py", line 313, in ProcessTurn
    print(AfterWordTiles(placedSection,userInputs.word,indexedCoor
d,userInputs.rotation))
  File "E:\Py\Scrabble\Documentation\Sub Systems\_WordPlacingBuild
ing.py", line 277, in AfterWordTiles
    while (section_in[indexPointer] != "+" and indexPointer < 15):
IndexError: list index out of range
```

Code breakdown

After programming the method, it seemed as though it should work. Yet, the function threw an `IndexError`. Dry running through the function, the while loop evaluates its expressions left to right (if there are more than one, like in my case). In other words, `section_in[indexPointer] != "+"` is evaluated first, then `indexPointer < 15` is evaluated, and then compared using the logical `and` operator. Basically, for this loop to determine if its condition is true (and thus loop) it has to unsafely evaluate the `section_in[indexPointer] != "+"` expression, so as the `indexPointer` increments it may exceed the list size of fifteen. Meaning that, if it throws an `IndexError`, the program is forcefully stopped.

From this I gathered that my loop condition was unsuitable, and I am not too sure why I thought it was a safe expression to evaluate outside of a try-except. Instead, of implementing a try-except statement I will instead change the loop condition, to simply loop if the `indexPointer` is less than the boundary index (15). Then within the loop, perform the conditional for an empty cell collision, if an empty cell is hit, break out of the loop.

Adjusted code

```
def AfterWordTiles(section_in,word_in,indexcoord_in,rotation_in):
    #Determine index required
    if (rotation_in): #Right placement
        indexPointer = indexcoord_in[0]+len(word_in)
    else: #Down placement
        indexPointer = indexcoord_in[1]+len(word_in)
    #If equal to 15/placement at border return empty
    if (indexPointer > 14):
        return ""
    charsAfter=[]
    #Loop if not at boundary
    while (indexPointer < 15):
        #If current cell is not empty
        if (section_in[indexPointer] != "+"):
            #Store value of current index in list
            charsAfter.append(section_in[indexPointer])
            #Increment index
            indexPointer+=1
        else: #If hit a empty cell
            break
    #Return as a string
    return ''.join(charsAfter)
```

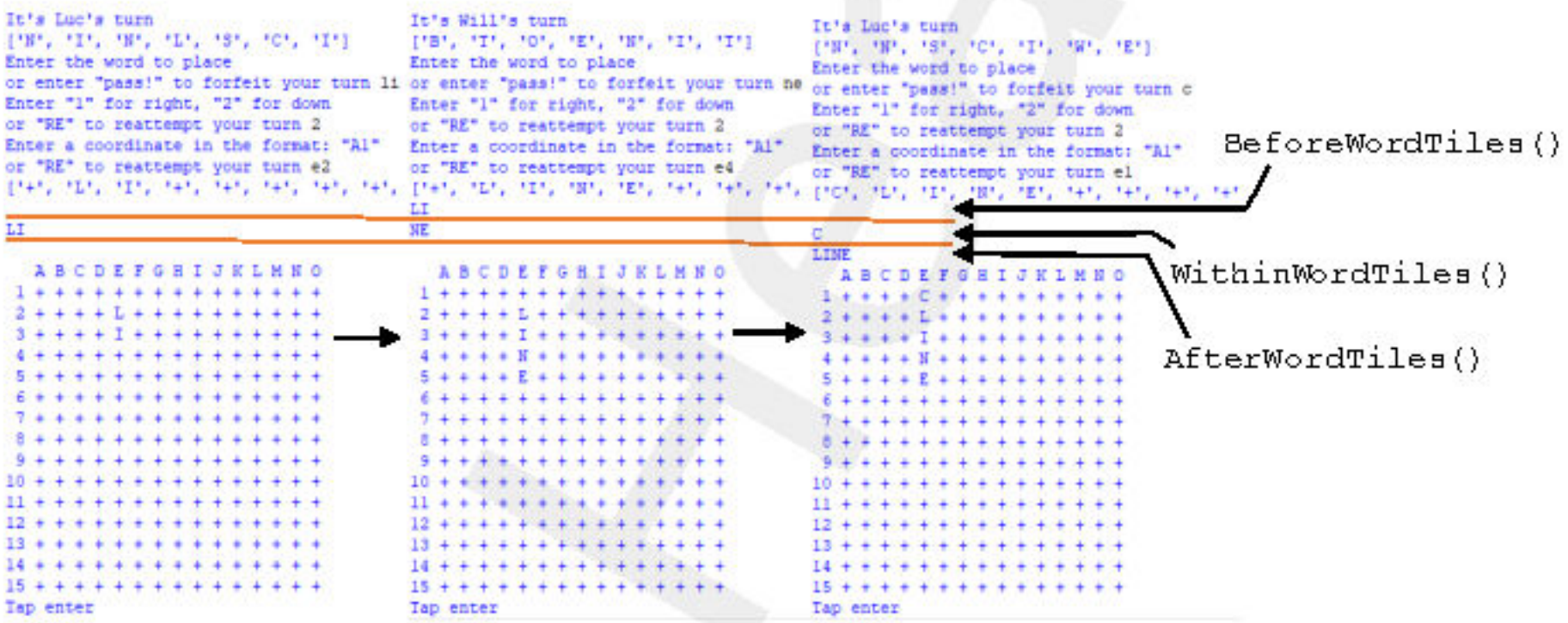
Adjusted code

```
def BeforeWordTiles(section_in,indexcoord_in,rotation_in):
    #Determine index required
    if (rotation_in): #Right placement
        indexPointer = indexcoord_in[0]-1
    else: #Down placement
        indexPointer = indexcoord_in[1]-1
    #If equal to -1/placement at border
    #return empty
    if (indexPointer < 0):
        return ""
    charsBehind=[]
    #Loop if not at boundary
    while (indexPointer >= 0):
        #If current cell is not empty
        if (section_in[indexPointer] != "+"):
            #Store value of current index in list
            charsBehind.append(section_in[indexPointer])
            #Decrement index
            indexPointer-=1
        else: #If hit empty cell
            break
    #Reverse the list and return as a string
    return ''.join(reversed(charsBehind))
```

Adjusted code breakdown

After making the modification to the loop for the `AfterWordTiles` function, I felt it was required to do the same for the `BeforeWordTiles` method, as they are so similar and the same error could occur.

Output – Usage of each validation method: Test one



Output – Usage of each validation method: Test two

Sub system(s) breakdown

Here you can see all three methods working as expected, including the newly developed `AfterWordTiles` function. All that is required for this stage of validation now, is to concatenate the three outputs, forming the fully constructed word, which can then be checked against a dictionary. The next large step will be the development of the surrounding word validation system, as currently this selection of validation methods is heavily dependent on the inputted direction from the user. For example, in the last code output from Test two, if a user were to enter a downward tile placement at the coordinate G9, such as the letter "T", the current validation is only used to construct the full main word, so it does not see the tile to the right of it ("O" at H9). Showing the requirement for surrounding word validation.

However, before beginning the development of it. I want to instead, implement the dictionary check, as well as a small method that verifies that the entered tiles from the player are actually within the player's inventory.

Sub system – Dictionary check

The first main requirement for this system to work is a dictionary file, that will be used to look-up valid English words. After a long search, looking at multiple sources, some that seemed to miss a large portion of known words. Finally, I came across a file that was large enough, containing most if not all words, the only slight issue was that it was in the JSON file format, yet it was formatted like a Python dictionary. With this, the program would have to read in the dictionary at the initialisation stage, then quickly lookup a string within the dictionary and determine if it is a valid English word.

Sample from the dictionary


```
{"a": 1, "aa": 1, "aaa": 1, "aah": 1, "aahed": 1, "aahing": 1, "aahs": 1, "aal": 1, "aalii": 1, "aaliis":
1, "aals": 1, "aam": 1, "aani": 1, "aardvark": 1, "aardvarks": 1, "aardwolf": 1, "aardwolves": 1, "aargh":
1, "aaron": 1, "aaronic": 1, "aaronical": 1, "aaronite": 1, "aaronitic": 1, "aarrgh": 1, "aarrghh": 1,
"aaru": 1, "aas": 1, "aasvogel": 1, "aasvogels": 1, "ab": 1, "aba": 1, "ababdeh": 1, "ababua": 1, "abac":
1, "abaca": 1, "abacay": 1, "abacas": 1, "abacate": 1, "abacaxi": 1, "abaci": 1, "abacinate": 1,
"abacination": 1, "abacisci": 1, "abaciscus": 1, "abacist": 1, "aback": 1, "abacli": 1, "abacot": 1,
"abacterial": 1, "abactinal": 1, "abactinally": 1, "abaction": 1, "abactor": 1, "abaculi": 1, "abaculus":
1, "abacus": 1, "abacuses": 1, "abada": 1, "abaddon": 1, "abadejo": 1, "abadengo": 1, "abadia": 1,
"abadite": 1, "abaff": 1, "abaft": 1, "abay": 1, "abayah": 1, "abaisance": 1, "abaised": 1, "abaiser": 1,
"abaisse": 1, "abaissed": 1, "abaka": 1, "abakas": 1, "abalation": 1, "abalienate": 1, "abalienated": 1,
"abalienating": 1, "abalienation": 1, "abalone": 1, "abalones": 1, "abama": 1, "abamp": 1, "abampere": 1,
"abamperes": 1, "abamps": 1, "aband": 1, "abandon": 1, "abandonable": 1, "abandoned": 1, "abandonedly": 1,
"abandonee": 1, "abandoner": 1, "abandoners": 1, "abandoning": 1, "abandonment": 1, "abandonments": 1,
"abandons": 1, "abandum": 1, "abanet": 1, "abanga": 1, "abanic": 1, "abannition": 1, "abantes": 1,
"abapical": 1, "abaptiston": 1, "abaptistum": 1, "abarambo": 1, "abaris": 1, "abarthrosis": 1,
"abarticular": 1, "abarticulation": 1, "abas": 1, "abase": 1, "abased": 1, "abasedly": 1, "abasedness": 1,
"abasement": 1, "abasements": 1, "abaser": 1, "abasers": 1, "abases": 1, "abasgi": 1, "abash": 1,
"abashed": 1, "abashedly": 1, "abashedness": 1, "abashes": 1, "abashing": 1, "abashless": 1, "abashlessly":
1, "abashment": 1, "abashments": 1, "abasia": 1, "abasias": 1, "abasic": 1, "abasing": 1, "abasio": 1,
"abask": 1, "abassi": 1, "abassin": 1, "abastard": 1, "abastardize": 1, "abastral": 1, "abatable": 1,
"abatage": 1, "abate": 1, "abated": 1, "abatement": 1, "abatements": 1, "abater": 1, "abaters": 1,
"abates": 1, "abatic": 1, "abating": 1, "abatis": 1, "abatised": 1, "abatises": 1, "abatjour": 1,
```


Development – Implementing a dictionary

To begin with, I must develop a method of reading in the file and convert it to a Python dictionary data type. Luckily Python has very simple file IO handling built-in, which will make it fairly simple to open the file, store the contents of it within a variable, then convert it to a dictionary usable by Pythons built-in methods.

Code

```
def CreateDictionary():
    #Read contents of file
    file = open("words_dictionary.json","r")
    #Literally convert to correct data type
    dictionary = ast.literal_eval(file.read())
    print("Dictionary build successful. Dictionary size:",len(dictionary))
    return dictionary
#Main program
dictionary = CreateDictionary()
...
```

Code breakdown

Using a very common file IO statement, open, it has a few different modes, specified as the second parameter, depending on the intended usage. For example, in this case the program only needs to read-in the contents of the file, so I can simple used "r". If the program required writing to a file, I could instead use "w".

Output

```
Dictionary build successful. Dictionary size: 370101
Enter the number of players
```

The second statement is a little more important, as it performs the main conversion, from a string in the structured format of a dictionary, to a dictionary data type that is usable by the program. For it to work, the program required another library named `ast`, which contains the `literal_eval` function, it essentially takes the given string (being the file contents) and treats it, as if it is a dictionary that has been converted to a string. In other words, if there was a string containing the syntax of a list, e.g. `"["A","B","C"]"`, this function will convert it from a string, to a Python list data type.

After converting the file contents to a Python dictionary, a little output is given, displaying the number of dictionary references are within this specific file, which turns out to be 370,101. Then the dictionary data type is returned to the `dictionary` variable, which can be used throughout the program.

Development – Comparing a word to the dictionary

With the dictionary loaded into the program, it is now time to develop the function that will return whether or not the given value is within the dictionary. A consideration I have made, is that this function will obviously require access to the dictionary file to perform its task, I can either pass the whole dictionary, along with the value to be checked whenever the function is required. Or I can use global variables. Neither is a good choice; however, one seems significantly more performant than the other. Technically, because Python parameters are passed by reference, it means that a 370,101-element dictionary is not being copied over and over, again, which is a good thing. However, it is still a pain to pass the dictionary through multiple functions before it is actually used. So, instead I will use a global variable for the dictionary. This means that the variable is accessible from anywhere within the program, at the cost of the variable never going out of scope, and causing the program to become slightly more confusing to follow.

The counterargument for using a global variable for this case. Is that this will be the only global variable within the program, and it should not affect the following/debugging of the program as it is will only be referenced once (within this function), so its usage will be very limited, where other functions only interface with this function, and do not access the dictionary directly.

Code

```
#Is string in dicitonary?
def IsValueInDictionary(value):
    #Using global variable dictionary
    #Dictionary is in lowercase
    if (value.lower() in dictionary):
        return True
    return False

#Main program
dictionary = CreateDictionary()
print(IsValueInDictionary("Avocado"))
...
```

Value to evaluate

Output

```
Dictionary build successful. Dictionary size: 370101
True
```

Code breakdown

It can be noted that the code is very simple, taking in a parameter, which should be a string. The function converts it to lowercase using the `lower` function attribute, as the dictionary is in lowercase. If the value is apparent return `True`, otherwise `False`.

Sub system – Owned tiles

Moving onto the second method that I wanted to quickly implement before continuing to the next major sub system (surrounding word validation). Determining, if a player owns the tiles required to produce the word placement they specified. This will have to consider if a player uses blanks during the turn. The solution to this problem, that I have in mind, is to create a copy of the player's inventory, and for each letter within the word, check if it exists within the player's inventory and remove it. Otherwise, check if the player possesses a blank tile. If so, assume that they want to use it for the letter that was not found within their inventory and remove it from the copy of the player's inventory. However, if the player does not possess the require tile for a placement, and does not possess a blank tile, the placement is invalid.

Development – Owned tiles

Firstly, this function will clearly require the current player's inventory as well as the word they requested to place. Then, using the `copy` library that was imported into the program during development of the `CopyGrid` function. Create a true copy of the player's inventory, which will be checked against. The function will then loop through the player's word request, which will allow the method to check if the letter is owned by the player.

If it is, the function will remove the first occurrence of the letter, using the `index` method, meaning it will not be detected more than once. But, if the player does not possess the letter, the function will then check for any blank tiles. If any are found, the first occurrence of a blank will be removed, and the loop will continue. In the

case that the player does not own a blank, and does not own the letter required, the function will return `False`. Because, as soon as one letter is not useable the whole placement cannot be made.

Code

```
def DoesPlayerOwnTiles(word_in,inventory_in):
    #Create a true copy of inventory
    tempInv = inventory_in.copy()
    #For each letter in word
    for x in word_in.upper():
        #If not using a skip character
        if (x != " "):
            #If player has tile
            if (x in tempInv):
                #Remove first occurrence of tile
                del tempInv[tempInv.index(x)]
            #Player doesn't have tile
            else:
                #If player has a blank
                if (" " in tempInv):
                    #Remove first occurrence of blank
                    print("Used blank for ",x)
                    del tempInv[tempInv.index(" ")]
                #Cannot make placement
                else:
                    print("You do not own ",x)
                    return False
    return True
inv = ["E","A","T","E","D"," "]
print(DoesPlayerOwnTiles("Eat",inv),end="\n\n")
print(DoesPlayerOwnTiles("tide",inv),end="\n\n")
print(DoesPlayerOwnTiles("SIDE",inv),end="\n\n")
print(DoesPlayerOwnTiles("teen",inv))
```

Code breakdown

Without the true copy of the player's inventory, it would be modified by the function, which is not the intended outcome.

The usage of the built-in `index` method, locates the first occurrence of a given string within a value.

The function exits on the first occurrence of a tile that the player does not own, or can use a blank for.

Using the useful `end` parameter, which is an argument of `print`, that allows you to specify what the output is escaped with, which by default is a new line. I have used it to format the outputs.

Outputs

```
True                    1

Used blank for  I       2
True

Used blank for  S       3
You do not own  I
False

Used blank for  N       4
True
```

Explained outputs

String	Output	Notes
Eat	True	Each letter is contained within the inventory given, so the loop if fully complete and True is returned.
tide	True Used blank for I	On the second iteration the letter I was not within the inventory. The program then looked for a blank and found one. Removing it, and continuing until the loop was complete returning True.
SIDE	Used blank for S You do not own I False	On the first iteration the letter S was not found within the inventory, in which the blank was used. However, on the second iteration, the letter I, was not found, and there was not a blank to replace it, so the function returned False.
teen	Used blank for N True	On the fourth iteration the letter N was not found, but a blank was apparent to replace it.

Sub system breakdown

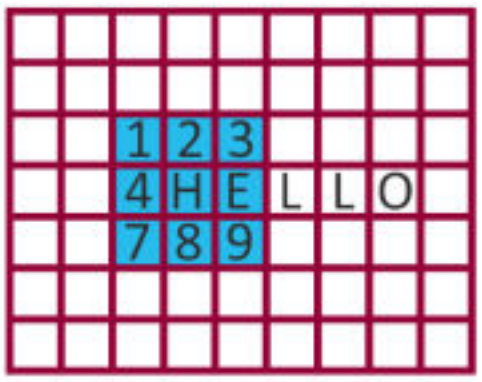
With this method developed, it can now be implemented within `ProcessTurn`. Detecting, if the word entered can be produced using the player's inventory, if so the system continues. Otherwise, allow the player to reattempt their inputs. It was simply added as an extra condition required before continuing the processing of the player's inputs, as shown here:

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    while True:
        #Gather user input
        userInputs = RetrieveInputs()
        #Handling returns
        if (userInputs == Event.SKIPTURN):
            print("Forfeiting turn")
            return
        elif (userInputs == Event.RESETTURN):
            print("Reattempting inputs")
        #If successful user return
        elif (type(userInputs) == InputData):
            #If word is constructable using their tiles
            if (DoesPlayerOwnTiles(userInputs.word,playerLetters_in)):
                break
            else:
                print("You do not have the tiles to make this word")
    #Continue processing
    ...
```

Sub system - Surrounding word validation

With the small implementations of the required features and some prerequisites made, it is time to develop the method of Surrounding word validation. At the beginning of development, my initial idea to produce this method, involved cycling through each letter in the player's placement, gathering cell data in a 3x3 area around each letter, which would have looked something similar to this:

The 3x3 method



For the placement of "HELLO", this method would require nine searches per letter, which for a five-letter word, would equal forty-five searches. However, this is an over-estimate as I could remove the searches at cells labelled 1,3,7 and 9, as they do not affect the letter H. Also, knowing that I already have a group of methods performing before, within and after checks. I could also remove the need for checks at cells 4,5 and 6, meaning for a horizontal placement only cells 2 and 8 would require searching. Bringing that maximum estimate down to 10 searches. In other words, for a horizontal placement, only the cells above and below each letter need to be searched, in-order to determine all the potential surrounding tiles.

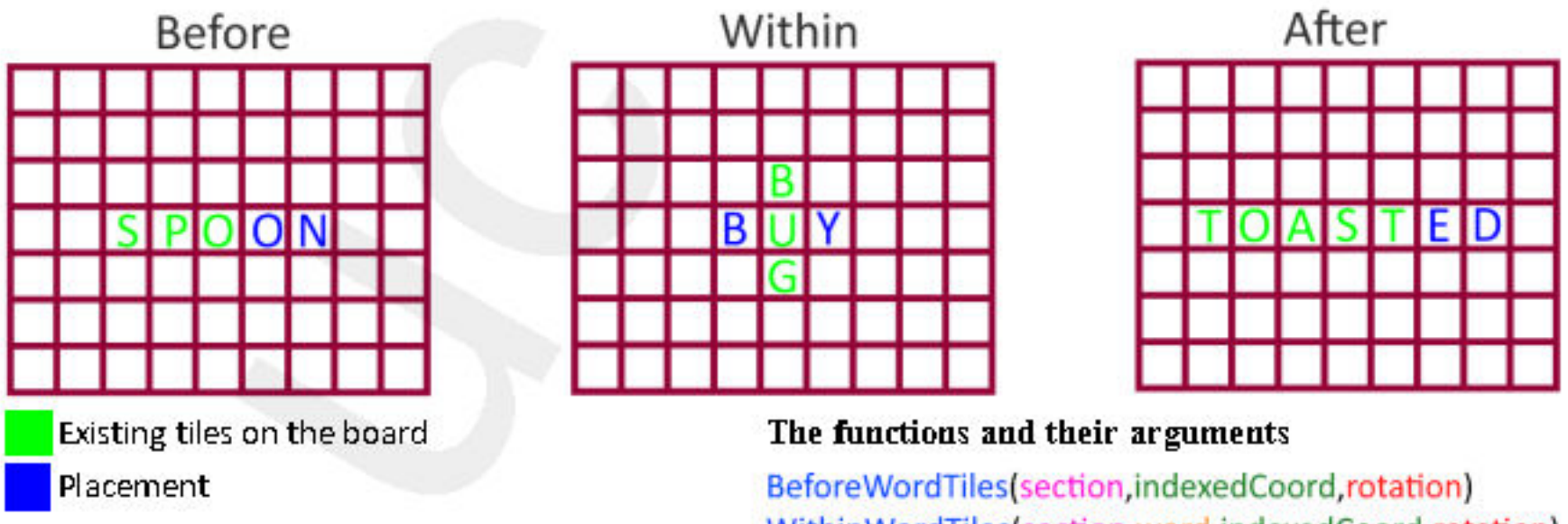
I could have developed this method, for detecting the potential tiles surrounding a user's placement. However, it seemed fairly over-complicated. Requiring a large amount of interaction with the two-dimensional array (game board), in addition to this, the code required for this method would be complex and confusing, as it would involve lots of iterative two-dimensional array traversal, requiring alot of comments and documentation, to explain. It would not get any simpler in the case that a surrounding tile was actually encountered. For example, if there were existing tiles at cells 2 and 8, I would then have to develop a method that reads upwards until an empty tile is collided with, concatenating the found tiles, to the current letter, and then to the tiles below cell 8, to discover the full connected word.

After considering the logic for searching above, and below a tile, it was then obvious, as to the method I was going to use to search for surrounding tiles.

The existing method

Due to the method in which I have designed and developed the systems for searching before, within and after a word placement, I can simply reuse them, treating each individual letter within a placement as the word to place, and using the opposite of the words rotation to search.

To more clearly illustrate my new method. I will re-illustrate the before, within and after validation methods.



The functions and their arguments

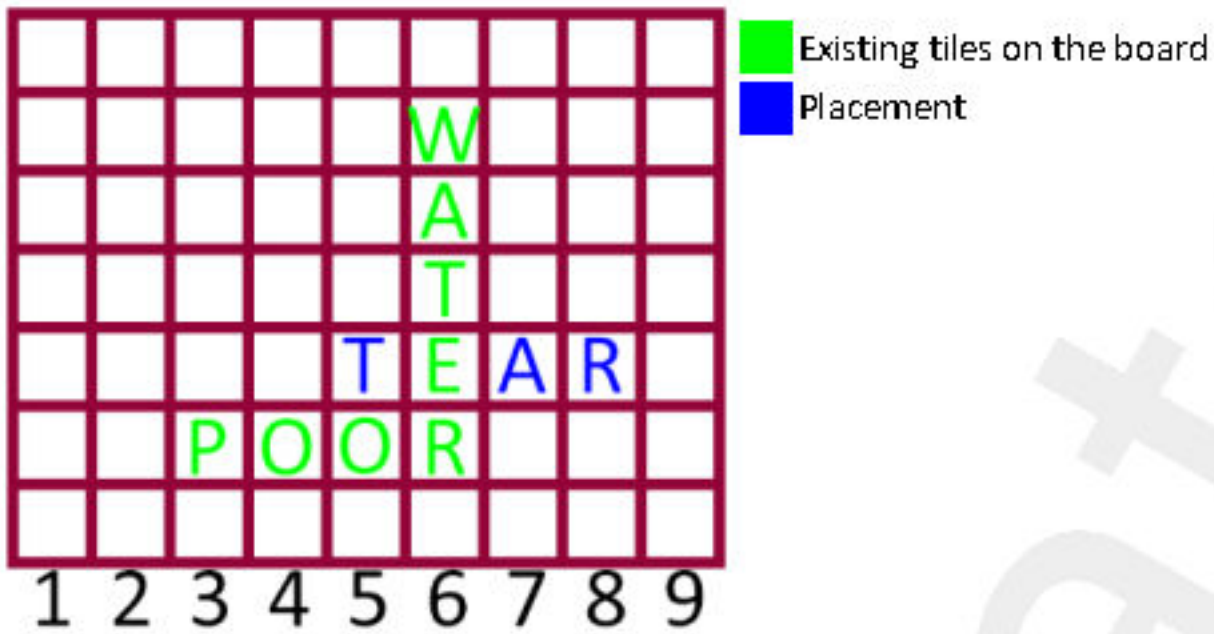
BeforeWordTiles(section,indexedCoord,rotation)
WithinWordTiles(section,word,indexedCoord,rotation)
AfterWordTiles(section,word,indexedCoord,rotation)

These methods perform their searches depending on the rotation that the player has inputted. For example, in the above illustrations each placement is horizontal, so each of the methods will search in the same direction to find the tiles that may exist: behind, within or after the placement.

Development – Surrounding word validation

Returning back to the method of searching for surrounding tiles, (for a horizontal placement) all the program needs to do. Is to treat each individual letter within the user's placement, as a placement in the opposite direction (a vertical placement), storing the found words as it cycles through each letter, which will then all be checked against a dictionary, in order for the placement to be detected as valid.

Illustration of the method, for a horizontal placement.



After the first set of validation checks have taken place (using the example above); searching for tiles before the placement (none), then within ("TEAR", finding the letter "E"), lastly after the placement (none). Giving the full concatenated word of "TEAR". This word is then added to a temporary list, containing all the words formed during this placement.

Once this has been completed, the Surrounding word validation method begins its execution. Firstly, it will loop through each letter within the player's inputted word ("T AR"). It will not loop through the full word ("TEAR") as the existing word ("WATER") containing the letter ("E"), does not require validating- as when it was placed, it will have gone through the same validation method. So, there is no reason to validated it again.

So, starting from the first character ("T"), the method will get the slice of the copy of the game board (sandbox grid), that is in the opposite direction to the word placement. In other words, the program will grab the column labelled "5", as it is the slice in the opposite direction to the placement. It will then pass this slice (as the section), along with the letter currently being searched (as the word), the index-based coordinate of the current letter being searched, and finally the opposite rotation of the user's placement.

The concatenated results of the three methods can then be stored within the temporary list, along with the full word, and the process repeats for all the letters within the placement.

Dry running through the example above, would output:

Column number	Before	Within	After	Word formed
5	""	T	O	TO
6	Skipped	Skipped	Skipped	Skipped
7	""	""	""	""
8	""	""	""	""

With the validation check finished. The temporary list of formed characters, contains: TEAR and TO. For the placement to be valid, all of the formed words must be valid English words, and so each element within the list is checked against the dictionary. Where in this example, both words would return `True`. Finally, the `ProcessTurn` method would continue and the word will be placed in the main game board. If not, the player's turn will be reset, allowing them to reattempt their turn.

To begin programming the method, I firstly need to encapsulate the function calls to the three validation checks, as well as the slicing method required, then returning the concatenated result. This will simplify the code used to gather the tiles in the direction given. The method will simply involve, gathering the section of the sandbox grid, and storing the result of the function calls, then finally, returning the concatenated values.

Code

```
#Search in direction for tiles
#Return formed words
def DirectionalGather(grid_in,word_in,rotation_in,coord_in):
    #Grab the section
    placedSection = GrabSection(grid_in,rotation_in,coord_in)
    #Gather data from each direction
    beforeTiles = BeforeWordTiles(placedSection,coord_in,rotation_in)
    withinTiles = WithinWordTiles(placedSection,word_in,coord_in,rotation_in)
    afterTiles = AfterWordTiles(placedSection,word_in,coord_in,rotation_in)
    return beforeTiles+withinTiles+afterTiles

def ProcessTurn(playerLetters_in, gameGridInUse):
    ...
    #Create copy of gamegrod
    sandboxGrid = CopyGrid(gameGridInUse)
    #Write word to sandbox grid
    PlaceWord(userInputs.rotation,userInputs.word,indexedCoord,sandboxGrid)
    #Debug full word output
    print(DirectionalGather(sandboxGrid,userInputs.word,userInputs.rotation,indexedCoord))
    ...
```

Output

```
It's Will's turn
['I', 'R', 'S', 'I', 'D', 'I', 'I']
Enter the word to place
or enter "pass!" to forfeit your turn si
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn e7
SIR
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + I + + + + + + + +
 6 + + + + + + Y + + + + + + + +
 7 + + + + S I R + + + + + + + +
 8 + + + + + + E + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
Tap enter
```

Now using this function, I can now program the surrounding tiles function. Using the `DirectionalGather` method, it will pass the sandbox game board to be sliced, the current letter within the player's input that is being iterated, the opposite of the placement's direction, as well as the index coordinate of the current letter, which will equal the starting position of the placement, plus an iterating value. That is equal to the index of the current letter within the word.

Code

```
def GrabAllFormedWords(grid_in,word_in, rotation_in,startpos_in):
    #Store each word formed
    formedWords = []
    #Store and grab main word
    formedWords.append(DirectionalGather(grid_in,word_in,rotation_in,startpos_in))
    #Create copy of the original startPos
    iteratingStartPos = startpos_in.copy()
    #Get the opposite of placement rotation
    if (rotation_in): #Right
        oppositeRot = False
    else:
        oppositeRot = True
    #For length of players input
    for letter in word_in.upper():
        #If player used a skip character, ignore
        #and iterate to next letter
        if (letter != " "):
            #Gather potential tiles at current letter
            wordFound = DirectionalGather(grid_in,letter,oppositeRot,iteratingStartPos)
            #Store result
            formedWords.append(wordFound)
        #Increment in direction
        if (oppositeRot):#If original is down
            iteratingStartPos[1]+=1
        else: #If original is right
            iteratingStartPos[0]+=1
    print(formedWords)
```

Code breakdown

The programming of this method was fairly simple in comparison to the undocumented thought process, behind the method. The reuse of pre-existing functions within the program simplified this method drastically. Running through the method, the function firstly declares an empty list, which will contain each formed word during the placement. Starting with the first element which is equal to the main placement, which is gathered by passing the values given by the user. Next, a copy is made, of the indexed coordinate given to the function, so that changes can be made to the variable, without modifying the original value. Because arguments are passed by reference and not by value. After this, the function stores a variable equal to the opposite direction given from the user (for the main placement), this is done so that, for a horizontal placement, the function checks for existing tiles vertically, and vice versa for a vertical placement. Then a for loop, iterates through each value within the player's inputted word, skipping skip characters, because they should be filled in with existing tiles from the grid, and because they do not require validating by this method as they will have already gone through this validation method, when they were placed. If the current letter is not a skip character, grab the result of a potential word in the opposite rotation to the placement, at the current letter. This works by passing the current letter, `letter`, as the `word_in` argument, for the `DirectionalGather` function, the position of the current letter within the grid, as well as the opposite rotation to the full word.

The result of this is then stored within `formedWords`. At the end of each iteration the correct axis within the copy of `startpos_in` (the index-based coordinate) is incremented by one, allowing the next character within the placement to be searched. Once this function completes its execution it will return the list containing all the formed words during the placement, which will then be checked against a dictionary.

Outputs

```
It's Luc's turn
['O', 'N', 'Z', 'O', 'O', 'O', 'S']
Enter the word to place
or enter "pass!" to forfeit your turn soon
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn j5
['SOON', 'S', 'O', 'O', 'N']
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + S + + + + +
 6 + + + + + + + + + O + + + + +
 7 + + + + + + + + + O + + + + +
 8 + + + + + + + + + N + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

1

```
It's Will's turn
['O', 'I', 'S', 'E', 'L', 'E', 'T']
Enter the word to place
or enter "pass!" to forfeit your turn se t
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn h8
['SENT', 'S', 'E', 'T']
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + S + + + + +
 6 + + + + + + + + + O + + + + +
 7 + + + + + + + + + O + + + + +
 8 + + + + + + + S E N T + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

2

```
It's Tom's turn
['G', 'D', 'R', 'I', ' ', 'I', 'B']
Enter the word to place
or enter "pass!" to forfeit your turn g t
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn i7
Used blank for  T
['GET', 'GO', 'T']
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + S + + + + +
 6 + + + + + + + + + O + + + + +
 7 + + + + + + + + G O + + + + +
 8 + + + + + + + S E N T + + + +
 9 + + + + + + + + T + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

3

Sub system breakdown

Here you can see the method working almost perfectly as expected. Storing the fully formed word at the first index within the list, then any subsequent words that were also formed around the placement. It can be best seen in the third output. Outputting the fully constructed word "GET", from the user's input of "G T" as the first element of the list, then the additionally constructed word "GO" at the first letter within the player's input. The search at the letter "E" is skipped, as it's an existing letter apart of the previous placement of "SENT", which has already been validated, meaning there is no need to validate it again. The reason that the function also returns unwanted formed words (the single character outputs) is because the `WithinWordTiles` function apart of the `DirectionalGather` returns the "word" passed to it, as well as connected tiles found. Causing each letter within the user's inputted word to be output, whether it's connected to another tile, or not.

Obviously, the program does not need these values within the collection of words formed. As they could disallow a perfectly valid placement. To prevent this, I can simply remove all the one letter elements within the final list. As it is impossible to produce a one letter word, because each placement must connect to another, meaning there must be a minimum of two characters within a word. However, currently there is no validation for a word to be connected to another, so clearly, currently, it is still possible to have a one letter word, but in future developments, this rule will be implemented, and this will not be an issue.

Issues to resolve

Overwriting tiles

Before continuing development on the few final systems, I would like to resolve a few issues within the program. Firstly, despite being mentioned very early on in development, placements are still able to overwrite existing tiles on the board, showing the overwriting of the letter "S", almost forming a word that best describes my approach to resolving the issue, until now.

```
It's Will's turn
['E', 'I', 'R', 'R', 'N', 'T', 'E']
Enter the word to place
or enter "pass!" to forfeit your turn rent
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn d4
['RENT', 'R', 'E', 'NIEVE', 'I']
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + R E N T + + + + + + + +
 5 + + + + + I + + + + + + + + +
 6 + + + + + E + + + + + + + + +
 7 + + + + + V + + + + + + + + +
 8 + + + + + E + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

```
def PlaceWord(orientation,word_in,startPos, grid_in):
    startPosTemp = startPos.copy()
    #Temp capitalise
    word_in=word_in.upper()
    #If placed to the right
    if (not orientation):
        #For each letter in the inputted word
        for x in word_in:
            if (x == " "): #When a space is read skip a cell
                startPosTemp[1]=startPosTemp[1]+1
            else:
                #Assign current letter to element within grid passed
                grid_in[startPosTemp[0]][startPosTemp[1]] = x
                #Increment position by one
                startPosTemp[1]=startPosTemp[1]+1
    #Opposite for downward
    else:
        for x in word_in:
            if (x == " "): #When a space is read skip a cell
                startPosTemp[0]=startPosTemp[0]+1
            else:
                grid_in[startPosTemp[0]][startPosTemp[1]] = x
                startPosTemp[0]=startPosTemp[0]+1
```

The ideal solution would be to add a conditional within the `PlaceWord` method, detecting for an overwrite. Then move the location for the method call, within the infinite loop. This would allow the `RetrieveInputs` method to be recalled if the condition occurred, however, it currently cannot be moved. Because it is dependent on the variables: `indexedCoord` and `sandboxGrid`. They could be moved as well. But, it would cause repeated code. I decided to stick with the repeated code method as it was the simplest (but not most elegant) solution to this problem.

Adjusted Code

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    while True:
        #Gather user input
        userInputs = RetrieveInputs()
        #Handling returns
        if (userInputs == Event.SKIPTURN):
            print("Forfeiting turn")
            return
        elif (userInputs == Event.RESETTURN):
            print("Reattempting inputs")
        #If successful user return
        elif (type(userInputs) == InputData):
            #If word is constructable using their tiles
            if (DoesPlayerOwnTiles(userInputs.word,playerLetters_in)):
                #Convert coordinate to index based coord
                indexedCoord = ConvertCoordToIndex(userInputs.coord)
                #Create copy of gamegrid
                sandboxGrid = CopyGrid(gameGridInUse)
                #If letter overwrites another
                if (PlaceWord(userInputs.rotation,userInputs.word,indexedCoord,sandboxGrid)):
                    break
                else:
                    print("You cannot overwrite tiles")
            else:
                print("You do not have the tiles to make this word")

    #Continue processing
    #Create copy of gamegrid
    sandboxGrid = CopyGrid(gameGridInUse)
    #Write word to sandbox grid
    PlaceWord(userInputs.rotation,userInputs.word,indexedCoord,sandboxGrid)
    #Grab all words formed during placement
    GrabAllFormedWords(sandboxGrid,userInputs.word, userInputs.rotation,indexedCoord)
    #Write to grid
    PlaceWord(userInputs.rotation,userInputs.word,indexedCoord,gameGridInUse)
    #Remove used tiles
    LetterDeductor(playerLetters_in,userInputs.word)
    #Re-distribute tiles
    Dealer(playerLetters_in)
```

```
def PlaceWord(orientation,word_in,startPos, grid_in):
    startPosTemp = startPos.copy()
    #Temp capitalise
    word_in=word_in.upper()
    #If placed to the right
    if (not orientation):
        #For each letter in the inputted word
        for x in word_in:
            if (x == " "): #When a space is read skip a cell
                startPosTemp[1]=startPosTemp[1]+1
            else:
                #If cell is empty
                if (grid_in[startPosTemp[0]][startPosTemp[1]] == "+"):
                    #Assign current letter to element within grid passed
                    grid_in[startPosTemp[0]][startPosTemp[1]] = x
                    #Increment position by one
                    startPosTemp[1]=startPosTemp[1]+1
                else:
                    return False
    #Opposite for downward
    else:
        for x in word_in:
            if (x == " "): #When a space is read skip a cell
                startPosTemp[0]=startPosTemp[0]+1
            else:
                if (grid_in[startPosTemp[0]][startPosTemp[1]] == "+"):
                    grid_in[startPosTemp[0]][startPosTemp[1]] = x
                    startPosTemp[0]=startPosTemp[0]+1
                else:
                    return False
    return True
```

Output

```
It's Luc's turn
['X', 'F', 'E', 'I', 'B', 'Z', ' ']
Enter the word to place
or enter "pass!" to forfeit your turn brief
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn f5
Used blank for  R
['BRIEF', 'B', 'R', 'I', 'E', 'F']
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + B + + + + + + + + +
 6 + + + + + R + + + + + + + + +
 7 + + + + + I + + + + + + + + +
 8 + + + + + E + + + + + + + + +
 9 + + + + + F + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
Tap enter

It's Will's turn
['B', 'C', 'I', 'W', 'O', 'I', 'T']
Enter the word to place
or enter "pass!" to forfeit your turn two
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn e9
You cannot overwrite tiles
Enter the word to place
or enter "pass!" to forfeit your turn
```

Code breakdown

It is definitely far from the ideal solution, requiring a copy of the game grid to created, twice or more per turn. Once to allow this new overwrite check, then again when the word is being tested for validity. However, it is an issue that needs to be resolved, and this is a solution. It can be seen in the output, that a suitable error message is given and the player is allowed to reattempt their turn.

Filtering words formed

As seen in the newly developed `GrabAllWords` function, the `wordsFormed` list contains each letter within the input, whether its connected to another tile, or not. This is due to the method in which the `WithinWordTiles` function works, so to adapt to this, a small filter is required. As explained when the issue was first encountered, I will simply remove all entries from the list that are one character in length, before the list is returned.

Code

```
def GrabAllFormedWords(grid_in,word_in, rotation_in,startpos_in):
    ...
    #Filter single character words
    filteredWords = []
    for entry in formedWords:
        #If entry has more than one character
        if (len(entry) > 1):
            filteredWords.append(entry)
    print(formedWords)
    print(filteredWords)
    return filteredWords
```

Outputs

```
It's Luc's turn
['L', 'L', 'E', 'U', 'Y', 'D', 'D']
Enter the word to place
or enter "pass!" to forfeit your turn duel
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn f4
['DUEL', 'D', 'U', 'E', 'L']
['DUEL']
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + D + + + + + + + + +
 5 + + + + + U + + + + + + + + +
 6 + + + + + E + + + + + + + + +
 7 + + + + + L + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

```
It's Tom's turn
['I', 'E', 'O', 'R', 'E', 'I', 'P']
Enter the word to place
or enter "pass!" to forfeit your turn r p
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn g3
['RIP', 'R', 'UP']
['RIP', 'UP']
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + R + + + + + + + +
 4 + + + + + D I N E + + + + + +
 5 + + + + + U P + + + + + + + +
 6 + + + + + E + + + + + + + + +
 7 + + + + + L + + + + + + + + +
 8 + + + + + + + + + + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

Breakdown

It can be seen within the outputs, comparing the unfiltered list (`formedWords`) with the filtered list (`filteredWords`). That the output no longer contains the unneeded single character outputs. Next, this list will be evaluated against the dictionary, finally determining if the placement is valid.

Dictionary checking

With the main issues resolved, and the list of formed words, filtered. The returned list can now be cycled through and checked against a dictionary, and if all the words are confirmed as valid English words, the program will then finally place the word in the main game board. However, the problem with this occurs, in the case that an invalid word is formed, as like every other stage, the user is able to reattempt their inputs when their placement fails. So, the same allowance must be made here. Causing the same issue encountered above, with overwriting tiles, with the solution involving repeated code.

Returning back to my original design for the `ProcessTurn` method, being a caller method for all the other systems required to perform a turn. To refine this definition, I am going to pull statements from it, replacing them with simple functions that perform the same task(s). Making the readability of the function much clearer, by using function calls with meaningful names.

Current code

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    while True:
        #Gather user input
        userInputs = RetrieveInputs()
        #Handling returns
        if (userInputs == Event.SKIPTURN):
            print("Forfeiting turn")
            return
        elif (userInputs == Event.RESETTURN):
            print("Reattempting inputs")
        #If successful user return
        elif (type(userInputs) == InputData):
            #If word is constructable using their tiles
            if (DoesPlayerOwnTiles(userInputs.word,playerLetters_in)):
                #Convert coordinate to index based coord
                indexedCoord = ConvertCoordToIndex(userInputs.coord)
                #Create copy of gamegrid
                sandboxGrid = CopyGrid(gameGridInUse)
                #If letter overwrites another
                if (PlaceWord(userInputs.rotation,userInputs.word,indexedCoord,sandboxGrid)):
                    break
                else:
                    print("You cannot overwrite tiles")
            else:
                print("You do not have the tiles to make this word")
```

To better illustrate what I mean. I would like to produce more functions like this one (`DoesPlayerOwnTiles`), as it's very clear in its function within the program, and it removes a large block of code from the `ProcessTurn` method.

In comparison to the code block and function call that detects if an overwrite occurs, where there are many function calls and variables being managed.

So, for this detection of overwriting tiles, as well as the implementation of dictionary checking. I will simply move the code blocks to their own function, similar to that of `DoesPlayerOwnTiles`.

Code

```
#Returns True if any tile overwrites another
def DoesPlacementOverwrite(grid_in,word_in,rot_in,coord_in):
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(coord_in)
    #Create copy of gamegrid
    tempGridCopy = CopyGrid(grid_in)
    #If word overwrites another
    if (not PlaceWord(rot_in,word_in,indexedCoord,tempGridCopy)):
        print("You cannot overwrite tiles")
        return True
    return False

#Given a list of words check if all of the
#words are within the dictionary
def AreFormedWordsValid(words_in):
    #Check each words is a true word
    for word in words_in:
        #If word not in dictionary
        if (not IsValueInDictionary(word)):
            print(word, " is not a valid word")
            return False
    return True
```

Code breakdown

By simply moving the dependencies for each method, such as the `indexedCoord`, and copy of the main game grid. I am able to move the whole check to a function with a meaningful name, simplifying the `ProcessTurn` method.

The same has been done for the implementation of the dictionary check for all the words formed, moving the processing to a function, that would otherwise be within the `ProcessTurn` method.

After these additions, the *ProcessTurn* method now looks like this:

Adjusted code

```
def ProcessTurn(playerLetters_in, gameGridInUse):
    while True:
        #Gather user input
        userInputs = RetrieveInputs()
        #Handling returns
        if (userInputs == Event.SKIPTURN):
            print("Forfeiting turn")
            return
        elif (userInputs == Event.RESETTURN):
            print("Reattempting inputs")
        #If successful user return
        elif (type(userInputs) == InputData):
            #If word is constructable using their tiles
            if (DoesPlayerOwnTiles(userInputs.word,playerLetters_in)):
                #If word doesn't overwrite another
                if (not DoesPlacementOverwrite(gameGridInUse,userInputs.word,userInputs.rotation,userInputs.coord)):
                    #Convert coordinate to index based coord
                    indexedCoord = ConvertCoordToIndex(userInputs.coord)
                    #Create copy of gamegrid
                    sandboxGrid = CopyGrid(gameGridInUse)
                    #Write word to sandbox grid
                    PlaceWord(userInputs.rotation,userInputs.word,indexedCoord,sandboxGrid)
                    #Grab all words formed during placement
                    wordsToCheck = GrabAllFormedWords(sandboxGrid,userInputs.word, userInputs.rotation,indexedCoord)
                    #Are all the words formed in the dictionary
                    if (AreFormedWordsValid(wordsToCheck)):
                        break
    #Process the valid placement
    ...
```

Old code

```
#Convert coordinate to index based coord
indexedCoord = ConvertCoordToIndex(userInputs.coord)
#Create copy of gamegrid
sandboxGrid = CopyGrid(gameGridInUse)
#If letter overwrites another
if (PlaceWord(userInputs.rotation,userInputs.word,indexedCoord,sandboxGrid)):
    break
else:
    print("You cannot overwrite tiles")
```

The improvement in readability is very clear. It is also a logical change, as the *ProcessTurn* method does not require all the extra variables required for each function call, so moving it to a closed function was a suitable change. However, it can still be seen, that there is still a large code block within the nested-if statements, which consists of all the required dependencies for gathering the formed words during a placement. The most logical change that could be made, would be moving the whole code block to a function that handles the gathering of formed words, as well as checking if they are valid English words. In a similar way to the changes made above.

Code

```
def AreFormedWordsValid(grid_in,word_in,rot_in,coord_in):
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(coord_in)
    #Create copy of gamegrid
    sandboxGrid = CopyGrid(grid_in)
    #Write word to sandbox grid
    PlaceWord(rot_in,word_in,indexedCoord,sandboxGrid)
    #Grab all words formed during placement
    wordsToCheck = GrabAllFormedWords(sandboxGrid,word_in, rot_in,indexedCoord)
    #Are all the words formed in the dictionary
    if (AreWordsInDictionary(wordsToCheck)):
        return True
    return False
def ProcessTurn(playerLetters_in, gameGridInUse):
    while True:
        #Gather user input
        userInputs = RetrieveInputs()
        #Handling returns
        if (userInputs == Event.SKIPTURN):
            print("Forfeiting turn")
            return
        elif (userInputs == Event.RESETTURN):
            print("Reattempting inputs")
        #If successful user return
        elif (type(userInputs) == InputData):
            #If word is constructable using their tiles
            if (DoesPlayerOwnTiles(userInputs.word,playerLetters_in)):
                #If word doesn't overwrite another
                if (not DoesPlacementOverwrite(gameGridInUse,userInputs.word,userInputs.rotation,userInputs.coord)):
                    #If all formed words are valid and English
                    if (AreFormedWordsValid(gameGridInUse,userInputs.word,userInputs.rotation,userInputs.coord)):
                        #Placement is valid
                        break
    #Process the valid placement
    ...
```

Output

```
  A B C D E F G H I J K L M N O
1 + + + + + + + + + + + + + + +
2 + + + + + + + + + + + + + + +
3 + + + + + + + + + + + + + + +
4 + + + + + + W + + + + + + + +
5 + + + + + + I + + + + + + + +
6 + + + + + + N + + + + + + + +
7 + + + + + + E + + + + + + + +
8 + + + + + + S + + + + + + + +
9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
Tap enter
It's Rob's turn
['Y', 'U', 'I', 'T', 'E', 'W', 'R']
Enter the word to place
or enter "pass!" to forfeit your turn riter
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn h4
You do not have  R  to form this word
Enter the word to place
or enter "pass!" to forfeit your turn yute
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn g6
You cannot overwrite tiles
Enter the word to place
or enter "pass!" to forfeit your turn write
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn h4
['WRITE', 'WW', 'IR', 'NI', 'ET', 'SE']
['WRITE', 'WW', 'IR', 'NI', 'ET', 'SE']
WW  is not a valid word
Enter the word to place
or enter "pass!" to forfeit your turn
```

After these changes, the *ProcessTurn* method is now much clearer, and only contains the necessary calls for it to perform its task. To confirm that each check still achieves its function, such as: the word being only comprised of tiles that they own, the word not overwriting any existing tiles and lastly that all the formed words are valid. I have tested the program, failing at each condition, which can be seen working correctly in the output messages.

Also, it can be noted that after each failure of the user's input. The program still allows them to reattempt their turn.

To avoid some confusion, I also adjusted the name of the dictionary checking function, which was originally named *AreFormedWordsValid* to *AreWordsInDictionary*. Using the original name, for the method that calls the surrounding word function, and checks the values against a dictionary.

Sub system - Scrabble rules

The next step in development is the implementation of the rules, that govern the method of play, in the original game of Scrabble. This will include: the requirement that each word is connected; by sharing at least one tile, the first placement must be more than two characters in length, and must pass through the centre cell. There are many other rules, such as: words can only be placed to the right and downward, not diagonally or backwards and placed tiled cannot be removed or shifted. However, they do not require implementing, as the current user interface prevents the action from being made.

Development – Passing through the centre

To begin with, the first, of the two rules that I will need to implement is, the first placement passing through the central cell. Also, the requirement that this first placement word, has a minimum of two characters. It is clear that there are two separate problems here, one involving the location of the word placement, with the second involving the length of the placement. Where both of them require the knowledge of the current round, as this is a check that will only be assessed on the first turn. Using a round counter, will mean that a conditional is required, detecting if it is the first round of the game. This will have to consider the ability to pass a turn, in which, naturally, this round counter variable would increment, no longer equating to the first round, and so the requirement for a placement to pass through the centre, would no longer work.

A simple solution to this would be to only increment this round counter, if a placement is made. That way, forfeiting a player's turn will have no effect.

Moving onto the actual validation, that one of the characters within the placement passes through the centre cell. I could append a condition to the `ProcessTurn` method, which will place the word in another copy of the grid, checking if one of the positions of the tiles within the placement is equal to the centre index-based coordinate (7,7 or H8). With regards to my initial concern about memory usage, when making copies of game grid. I have noticed that the impact is almost negligible, with modern computing system.

Profiling memory

Using an interesting external module called `memory_profiler`, I was able to review memory usage of each function, using function decorators. I setup a short program that includes the `CopyGrid` function, using the same sized game grid, to evaluate its memory usage.

Code

```
import copy
from memory_profiler import profile

#Returns a copy of the grid passed
def CopyGrid(grid_in):
    return copy.deepcopy(grid_in)
@profile
#Simulate grid usage
def GridUsage(grid_in):
    storage = []
    for x in range(2):
        storage.append(CopyGrid(grid_in))
@profile
def Process():
    #Create grid
    gameGrid=[["+"]*15 for i in range(15)]
    #Use grid
    GridUsage(gameGrid)
#Main
Process()
```

The function decorator that is used by the module, to determine which functions will be profiled for memory usage.

Simulated usage. Generating a number of copies of the game grid and storing them.

With only two copies of the game grid in the scope of the `GridUsage` method, and one copy within the Process method, the memory usage outputted in a clean table is as follows:

```
Filename: E:\Py\Scrabble\Testing\memory usage.py

Line #    Mem usage    Increment   Line Contents
================================================
     7     25.4 MiB     25.4 MiB   @profile
     8                             #Simulate grid usage
     9                             def GridUsage(grid_in):
    10     25.4 MiB      0.0 MiB       storage = []
    11     25.4 MiB      0.0 MiB       for x in range(2):
    12     25.4 MiB      0.0 MiB           storage.append(CopyGrid(grid_in))

Filename: E:\Py\Scrabble\Testing\memory usage.py

Line #    Mem usage    Increment   Line Contents
================================================
    13     25.4 MiB     25.4 MiB   @profile
    14                             def Process():
    15                                 #Create grid
    16     25.4 MiB      0.0 MiB       gameGrid=[["+"]*15 for i in range(15)]
    17                                 #Use grid
    18     25.5 MiB      0.0 MiB       GridUsage(gameGrid)
```

To explain the output. Each separate table represents the execution of a function marked with the `@profile` decorator. So, the first table displays the memory usage of the `GridUsage` procedure. Showing that initially the execution of the program is allocated ~25 MB, regardless of the code.

Then for each line executed there is less than a MiB (1E6 bytes) allocated. Which is understandable for a two-dimensional list of this size, as the output is in a much larger scale, explaining why, 0.0 MiB is displayed.

However, by adjusting the number of iterations within `GridUsage`. It can be noted that, the number of copies of this grid must be fairly large, before any significant allocations are made.

For 200 iterations:

```
Filename: E:\Py\Scrabble\Testing\memory usage.py

Line #    Mem usage    Increment   Line Contents
================================================
     8     25.5 MiB     25.5 MiB   @profile
     9                             #Simulate grid usage
    10                             def GridUsage(grid_in):
    11     25.5 MiB      0.0 MiB       storage = []
    12     26.1 MiB      0.0 MiB       for x in range(200):
    13     26.1 MiB      0.1 MiB           storage.append(CopyGrid(grid_in))

Filename: E:\Py\Scrabble\Testing\memory usage.py

Line #    Mem usage    Increment   Line Contents
================================================
    14     25.5 MiB     25.5 MiB   @profile
    15                             def Process():
    16                                 #Create grid
    17     25.5 MiB      0.0 MiB       gameGrid=[["+"]*15 for i in range(15)]
    18                                 #Use grid
    19     26.1 MiB      0.6 MiB       GridUsage(gameGrid)
```

With this many iterations, the stack allocations start to become noticeable. Using about ~600 kB with 200 copies of the data structure.

Remembering, that in the Scrabble use case, only two copies (at max) of the two-dimensional list are within scope at any time. Which corresponds to very negligible usage in memory, especially within modern computing. Leading back to the target audience, which a reasonable assumption, is that their devices will not be impacted by the relatively few copies of the two-dimensional list that are made.

Resuming development – Passing through the centre

Regarding the method for checking whether the first placement passes through the centre cell, I will implement a round counter, which is incremented after each successful placement. As well as a function that (for the first placement of the game), checks the length of the word given, places the word in copy of the game board, and checks if the element at the central position is equal to an empty character.

Code

```
def DoesFirstWordMeetRequirements(grid_in,word_in,rot_in,coord_in):
    #Length check
    if (len(word_in) < 2):
        print("The first word must be 2 or more characters in length")
        return False
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(coord_in)
    #Create copy of gamegrid
    sandboxGrid = CopyGrid(grid_in)
    #Write word to sandbox grid
    PlaceWord(rot_in,word_in,indexedCoord,sandboxGrid)
    #Check if centre cell is populated
    if (sandboxGrid[7][7] == "+"):
        print("The first word must pass through the centre")
        return False
    return True

def ProcessTurn(playerLetters_in, gameGridInUse, roundcounter):
    ...
        #If successful user return
        elif (type(userInputs) == InputData):
            #If first placement
            if (roundcounter == 0):
                #If word isnt correct length and doesnt pass through centre
                if (not DoesFirstWordMeetRequirements(gameGridInUse,userInputs.word,userInputs.rotation,userInputs.coord)):
                    continue #Iterate loop again
            #If word is constructable using their tiles
            if (DoesPlayerOwnTiles(userInputs.word,playerLetters_in)):
                #If word doesn't overwrite another
                if (not DoesPlacementOverwrite(gameGridInUse,userInputs.word,userInputs.rotation,userInputs.coord)):
                    #If all formed words are valid and English
                    if (AreFormedWordsValid(gameGridInUse,userInputs.word,userInputs.rotation,userInputs.coord)):
                        #Placement is valid
                        break
    ...
    return True
#Game Loop
    ...
    #Call factory manager method, passing current players inventory
    #If placement occurs increment round
    if (ProcessTurn(playerInventories[currentTurn],gameGrid,roundCounter)):
        roundCounter += 1
    ...
```

Code breakdown

The rudimentary solution, checking that the length is greater than two, then placing the word in a copy of the grid, then checking if the central cell has been replaced by a character.

Also, the addition of the `roundCounter` variable within the main program, which is passed to the `ProcessTurn` method.

To determine if a placement occurs, I have added a return of True, which is detected in the main program, incrementing the `roundCounter`.

Output

```
It's Luc's turn
['T', 'B', 'I', 'O', 'E', 'W', 'U']
Enter the word to place
or enter "pass!" to forfeit your turn bite
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn 12
The first word must pass through the centre
Enter the word to place
or enter "pass!" to forfeit your turn bite
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn f8
['BITE', 'B', 'I', 'T', 'E']
['BITE']
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + B I T E + + + + + +
 9 + + + + + + + + + + + + + + +
10 + + + + + + + + + + + + + + +
11 + + + + + + + + + + + + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
```

Breakdown

Testing the code shows that, for any initial placement, that does not pass through the central cell, an error message will be displayed. The player's turn is then reset, allowing them to try again. There is not much to evaluate about this very small addition to the program, as there are only two variations to the placement that could occur here (being a downward/rightward placement).

Development – Connected words

For a placement to be valid (after the first placement), the word must share an existing tile on the board. For this to occur the player may use a skip character, or may place their word adjacent to another, forming additional words. A method of detecting that each subsequent placement is connected, would be to implement a check within the `AreFormedWordsValid` function. Firstly, if the placement forms more than one word (length of returned list `wordsToCheck` is greater than one), that means that it must be connected to another tile. Or, if the fully formed word (the first element within the `wordsToCheck` list) is no longer equivalent to the word entered by the user, as this would mean the player has used an existing tile from the board.

This method would work because, it firstly checks if the placement is connected via surrounding tiles, as it would form extra words. Or secondly, if the fully formed word is no longer equal to the word that was originally entered by the user, as that would mean, that existing tiles were found in-front, behind or within the word, meaning that it is connected.

Code

```python
def AreFormedWordsValid(grid_in,word_in,rot_in,coord_in,roundcounter_in):
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(coord_in)
    #Create copy of gamegrid
    sandboxGrid = CopyGrid(grid_in)
    #Write word to sandbox grid
    PlaceWord(rot_in,word_in,indexedCoord,sandboxGrid)
    #Grab all words formed during placement
    wordsToCheck = GrabAllFormedWords(sandboxGrid,word_in, rot_in,indexedCoord)
    #If not first placement
    if (roundcounter_in != 0):
        #If the placement is not connected to another
        if (len(wordsToCheck) == 1 and wordsToCheck[0].upper() == word_in.upper()):
            print("Word must connect to another")
            return False
    #Are all the words formed in the dictionary
    if (AreWordsInDictionary(wordsToCheck)):
        return True
    return False
```

Code breakdown

Using the logic described above, I have added an additional set of statements within the `AreFormedWordsValid` function. Firstly, determining if it is a subsequent turn, not the first. As it would not be possible to connect the first placement of the game, to any other word.

Secondly, a long statement, determining if there is only one formed word (the main word), *and* that the user-inputted word is identical to the only formed word. If both of these expressions are `True`, this would mean that the word is not connected to another. Thus, the placement is invalid.

Luc Heaton

Outputs

```
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + D R I V E + + +
 9 + + + + + + + + + + I + + + +
10 + + + + + + + + + + N + + + +
11 + + + + + + + + + + E + + + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
Tap enter
It's Luc's turn
['T', 'I', 'E', 'O', 'S', 'E', 'F']
Enter the word to place
or enter "pass!" to forfeit your turn feet
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn b2
['FEET']
Word must connect to another
Enter the word to place
or enter "pass!" to forfeit your turn
```

```
Enter the word to place
or enter "pass!" to forfeit your turn f et
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn j11
['FEET']
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + D R I V E + + +
 9 + + + + + + + + + + I + + + +
10 + + + + + + + + + + N + + + +
11 + + + + + + + + + + F E E T + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
Tap enter
```

```
   A B C D E F G H I J K L M N O
 1 + + + + + + + + + + + + + + +
 2 + + + + + + + + + + + + + + +
 3 + + + + + + + + + + + + + + +
 4 + + + + + + + + + + + + + + +
 5 + + + + + + + + + + + + + + +
 6 + + + + + + + + + + + + + + +
 7 + + + + + + + + + + + + + + +
 8 + + + + + + + D R I V E + + +
 9 + + + + + + + + + + I + + + +
10 + + + + + + + + + + N + + + +
11 + + + + + + + + + + F E E T + +
12 + + + + + + + + + + + + + + +
13 + + + + + + + + + + + + + + +
14 + + + + + + + + + + + + + + +
15 + + + + + + + + + + + + + + +
Tap enter
It's Will's turn
['C', 'H', 'G', 'K', 'E', 'T', 'I']
Enter the word to place
or enter "pass!" to forfeit your turn kite
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn h13
['KITE']
Word must connect to another
Enter the word to place
or enter "pass!" to forfeit your turn
```

Breakdown

With this implementation, each subsequent word is forced to connect to another, in-order to become a valid placement. This can be seen with the outputs. Firstly, showing that for a vertical placement of "feet", starting from the coordinate, B2, returns the error message, as the word is not connected to another.

Dry running through the code, as well as the debug message for the returned list of formed words. It can be seen that the list only contains one element, which meets the first condition, before being invalidated (other than being a subsequent turn). Secondly, comparing that single formed word, within the list, to the player's inputted word, meets the second condition, as the player's inputted word is identical to the formed word within the list. Meaning that the placement does not connect to another.

Sub system breakdown

With both rules fully implemented and performing their tasks as expected. Enforcing the rule that the first placement must pass through the central tile, as well as a condition that prevents unconnected words from being placed. Providing an adequate error message, while also allowing the user to reattempt their turn. After this, it is now time to begin the development of Bonus tiles. A method that will allow the detection of placements that overwrite specific cells on the board, which will then be used in addition to a scoring system, to determine a placements score.

Sub system – Bonus tiles

Following the Gantt chart that was produced, the implementation of Bonus tiles is now required. The main goal of this method is to display cells on the board that represent a bonus or multiplier for a placement/tile at a predefined position on the grid. This will include, as mentioned in the design:

-TWS, Triple Word Score
-DWS, Double Word Score
-TLS, Triple Letter Score
-DLS, Double Letter Score

The meanings of each bonus are not too relevant at this stage, as this system will only involve the displaying and distinguishing of bonuses, for the user and the program. Allowing the user to visually see, where a placement can be made to score them extra points, as well as, allowing the program to determine which bonus to apply. With the text-based display that I am using, the most sensible method of illustrating these bonus tiles will be using a special character for the four different bonus types. The reasoning behind using a single character instead of the abbreviated TLS or DWS, for example, is that the cells within the grid are all one character in size, so using anything greater than one character will shift the grid entirely, breaking the display

Regarding the characters that will be used, the requirements are: that it is not an alphabetic character, that can be placed by the user, and it must be a single character, so that the grid display is not broken. With this in mind I settled for these:

- 6, Triple Word Score
- 2, Double Word Score
- 3, Triple Letter Score
- 1, Double Letter Score

Obviously, the characters are not ideal, given that they are not very intuitive. Although, I did consider using Unicode characters, most of them are unaffected by the font used within the Python IDLE, and so they are not fixed-sized, causing the grid display to break. However, I tried to select slightly representative values, using some abstraction and trying to find some meaning of the integers used.

As a solution this, and to any features that require some explanation, I will develop a simple key/legend, listed alongside the game grid, which will give the explanations required and will occupy the large amount of empty space to the right of the grid. Yet, this idea will be postponed, until the base game has been completed, as it is of lower priority.

Continuing with the bonus tiles, for the technical design of this method. I briefly described within the Gantt chart that this method would, "setup bonus tiles in defined locations within the game grid", this would imply that the game grid that contains all the placement data, would also contain the data required for locating bonus tiles. This could be the method used to produce this sub system, however it would cause multiple issues within the program, such as the method in which I detect empty tiles, as it searches for the occurrence of "+"'s. Instead of this, I will implement a second grid, separate of the main game grid, that will contain the predefined locations of the bonus tiles, which will be used as an overlay, on-top of the main game grid.

Development – Bonus tiles

This would work by hardcoding a grid of the same size (15x15), which would contain the special characters representing the different bonus types, at their specific locations, with all the other elements blank. Then when drawing the main game board, using the `DrawGrid` method, also, read the contents of the bonus game grid, displaying the contents of the main game board, where there is a blank within the bonus game grid. But, displaying the special characters in-place of cells on the main game board.

Also, due to the fact that a bonus tile can only be used once. Using this method (above) would allow the bonus tile(s), when used, to be set as blank (within the bonus grid), meaning that they will no-longer overlay the tiles that would now appear below them.

Code

```
#Main program
bonusGrid=[["6", " ", " ","1", " ", " ", " ","6", " ", " ", " ","1", " ", " ","6"],
           [" ","2", " ", " ", " ", " ","3", " ", " ", " ", " ","3", " ", " ", " ","2", " "],
           [" ", " ","2", " ", " ", " ", " ","1", " ","1", " ", " ", " ","2", " ", " "],
           ["1", " ", " ", " ","2", " ", " ", " ", " ","1", " ", " ", " ", " ","2", " ", " ", " ","1"],
           [" ", " ", " ", " ", " ","2", " ", " ", " ", " ", " ", " ", " ","2", " ", " ", " ", " "],
           [" ","3", " ", " ", " ", " ","3", " ", " ", " ", " ","3", " ", " ", " ", " ","3", " "],
           [" ", " ", " ","1", " ", " ", " ", " ","1", " ", " ","1", " ", " ", " ", " ","1", " ", " ", " "],
           ["6", " ", " ", " ","1", " ", " ", " ", " ","0", " ", " ", " ", " ","1", " ", " ", " ","6"],
           [" ", " ", " ","1", " ", " ", " ", " ","1", " ", " ","1", " ", " ", " ", " ","1", " ", " ", " "],
           [" ","3", " ", " ", " ", " ","3", " ", " ", " ", " ","3", " ", " ", " ", " ","3", " "],
           [" ", " ", " ", " ", " ","2", " ", " ", " ", " ", " ", " ", " ","2", " ", " ", " ", " ", " "],
           ["1", " ", " ", " ","2", " ", " ", " ", " ","1", " ", " ", " ", " ","2", " ", " ", " ","1"],
           [" ", " ", " ","2", " ", " ", " ", " ","1", " ", " ","1", " ", " ", " ", " ","2", " ", " ", " "],
           [" ","2", " ", " ", " ", " ","3", " ", " ", " ", " ","3", " ", " ", " ", " ","2", " "],
           ["6", " ", " ", " ","1", " ", " ", " ", " ","6", " ", " ", " ", " ","1", " ", " ", " ","6"]]
```

Code breakdown

To begin with, I have predefined the 15x15 grid, containing the bonus tiles, as well as the blanks. Also, as an aside, when testing the game, I felt as-though I have to search for the central cell, to make this clearer. I will also mark it with a separate character. Compared to the grid of "+"'s, I will instead use a "0", as it is not a character that can be used within the game.

Using the traditional layout of bonus tiles.

The second part to this implementation, is the adaptation of the `DrawGrid` method, which will print the contents of the main grid, where the bonus grid contains a blank, and vice versa for when a bonus tile is found.

Code

```
def DrawGrid(grid_in,bonus_in):
    #Output top row coords
    print("   A B C D E F G H I J K L M N O", end = "")
    #For each item in board
    for y in range(15):
        for x in range(15):
            if (x % 15 == 0): #If at end of the row
                print("") #Start new line
                print("%2d" % (y+1,), end = " ")
            #If bonus contents is blank
            if (bonus_in[x][y] == " "):
                #Print game grid
                print(grid_in[x][y], end = " ")
            else: #If bonus contents is populated
                #Print special character
                print(bonus_in[x][y], end = " ")
    print("") #Start new line
```

Output

```
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + 0 + + + 1 + + 6
 9 + + 1 + + + 1 + 1 + + + 1 + +
10 + 3 + + + 3 + + + 3 + + + 3 +
11 + + + + 2 + + + + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

Code breakdown

After adding the required parameter, to allow the passing of the bonus grid. I've added an extra condition determining if the contents of the current position in the bonus grid is empty, if so print the element at the same position within the main game board. Otherwise, print the special character, giving this result.

The problem with this addition, is if a placement intercepts one of the bonus tiles. The placement will occur as usual; however, the bonus character will be displayed instead of the tile that was placed. The solution to this, and the final part of this implementation involves the understanding of the usage of bonus tiles.

Bonus tiles explained

Starting with the operation of each type of bonus:

- Triple Word Score - The individual tile scores of the fully formed word is summed, and multiplied by three.
- Double Word Score - The individual tile scores of the fully formed word is summed, and multiplied by two.
- Triple Letter Score - The score value of the tile that lies on the bonus cell is multiplied by three.
- Double Letter Score - The score value of the tile that lies on the bonus cell is multiplied by two.

Due to the layout of the bonus tiles, and the nature of combining tiles in this game, it is possible to form a single word that spans the length of the board. This means that it is possible for a single placement to use more than one bonus tile, for example, a word that is five letters in length, and placed horizontally at B2. Will require the last tile's score to be tripled, plus the sum of all the other tiles. It is also worth noting that the letter bonuses are applied first, then the word bonuses, otherwise the scoring would be inconsistent. In addition to this, the bonus tile's effect only applies to the main placement word, it is not applied to other formed words during the placement.

Development – Detecting bonus tiles

With this in-mind, I must firstly develop a function that searches at the cell positions of the user's placement, within the bonus grid, that will determine which bonuses were intercepted. Also, removing them from the bonus grid, so that they are no applied more than once, and so that they no-longer overwrite the placed word.

Code

```
def DetectBonusUsage(bonusgrid_in,word_in,rot_in,coord_in):
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(coord_in)
    #Store each component of the coord
    rowIndex = indexedCoord[1]
    colIndex = indexedCoord[0]
    #Iterate for each letter within input
    for x in range(len(word_in)):
        #Detect and remove centre marker
        if (bonusgrid_in[rowIndex][colIndex] == "0"):
            bonusgrid_in[rowIndex][colIndex] = " "
        #Detect and remove bonus type
        if (bonusgrid_in[rowIndex][colIndex] != " "):
            print("Bonus used:",bonusgrid_in[rowIndex][colIndex])
            bonusgrid_in[rowIndex][colIndex] = " "
        #Increment correct index
        if (rot_in): #Right
            colIndex+=1
        else: #Down
            rowIndex+=1
```

Code breakdown

Store components to make their usage simpler, rather than referencing to the indexes of `indexedCoord`.

Special case for central character, as it does not need to be accounted for. However, it needs to be removed once the player has made a placement through the centre.

Read the remove the bonus tile that may have been used in a placement, currently outputting the result.

Increment the correct axis depending on the orientation of the placement.

Despite this being a simple method, I spent a large amount of time trying to simplify the code, as well as remove most duplicate code, to reduce the line count. The next step for this method will be to determine the bonus that is applied, and in the case of a letter bonus, apply the bonus to the specific letter that landed on the bonus.

Outputs

```
It's Luc's turn
['L', 'U', 'E', 'I', 'T', 'I', 'T']
Enter the word to place
or enter "pass!" to forfeit your turn tile
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn h8
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + T I L E 1 + + 6
 9 + + 1 + + + 1 + 1 + + + 1 + +
10 + 3 + + + 3 + + + 3 + + + 3 +
11 + + + + 2 + + + + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

```
It's Will's turn
['E', 'M', 'T', 'R', 'D', 'W', 'E']
Enter the word to place
or enter "pass!" to forfeit your turn m tre
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn k7
Bonus used: 2
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + M + 1 + +
 8 6 + + 1 + + + T I L E 1 + + 6
 9 + + 1 + + + 1 + 1 + T + 1 + +
10 + 3 + + + 3 + + + 3 R + + 3 +
11 + + + + 2 + + + + + E + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

```
It's Tom's turn
['P', 'R', 'O', 'I', 'I', 'E', 'S']
Enter the word to place
or enter "pass!" to forfeit your turn so e
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn i10
Bonus used: 3
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + M + 1 + +
 8 6 + + 1 + + + T I L E 1 + + 6
 9 + + 1 + + + 1 + 1 + T + 1 + +
10 + 3 + + + 3 + + S O R E + 3 +
11 + + + + 2 + + + + + E + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

```
It's Luc's turn
['U', 'I', 'T', 'B', 'V', ' ', 'I']
Enter the word to place
or enter "pass!" to forfeit your turn b n
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn i7
Used blank for  N
Bonus used: 1
Bonus used: 1
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + B + M + 1 + +
 8 6 + + 1 + + + T I L E 1 + + 6
 9 + + 1 + + + 1 + N + T + 1 + +
10 + 3 + + + 3 + + S O R E + 3 +
11 + + + + 2 + + + + + E + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

Breakdown

Analysing the outputs in chronological order, firstly it can be seen that the centre marker "0" has been detected and removed from the bonus grid, with no further processing required. Next, the placement of "M TRE", to form the word "METRE", where the last character in the placement lands on the Double Word Score bonus, represented by a "2", which is shown in the debug message. Continuing, a horizontal placement is then made, where the second character lands on a Triple Letter Score, which is represented by a 3. Finally, in the last output it can be noted that the method works as expected for the usage of more than one bonus. Being that of two Double Letter Scores.

Sub system - Scoring

This is the point where the merging of scoring and bonus handling must occur, as to calculate a players score at the end of a placement, the potential usage of bonuses is required, as well as the individual score values assigned to each letter.

To begin with, I need to incorporate the score value that is associated with each tile, this will involve taking the score values that are used in the original game, and referencing them when used. Secondly, the final calculation of a players score at the end of a placement needs a clear definition, so:

Scenario 1. A placement is made, with no connections, with no usage of bonus tiles.
Score calculation. The sum of each individual tile score, used during the placement.
Scenario 2. A placement is made, with a connection(s), with no usage of bonus tiles.
Score calculation. The sum of each individual tile score, including the existing tile used, during the placement.
Scenario 3. A placement is made, with the usage of bonus tiles.

Score calculation. Firstly, Double/Triple Letter bonuses are calculated for the tiles that landed on them, this and the rest of the tiles that may have been used during the placement is summed, and if any Double/Triple Word bonuses are used, they are then applied.

Bonus type key

-Triple Word Score - TWS – 6
-Double Word Score - DWS – 2
-Triple Letter Score - TLS - 3
-Double Letter Score - DLS – 1

Here is an example of the calculation when multiple bonuses are used:

Placement word: H,E,L,L,O

Letter tile scores (in-order): 4,1,1,1,1

In this example these bonuses are used:
- The letter H lands on a Triple Letter bonus.
- The letter L lands on a Double Word bonus.
- The letter O lands on a Double Letter bonus.

Calculation:

2 x ((4 x 3) + 1 + 1 + 1 + (1 x 2)) = 34

Analysis:

Firstly, the letter bonuses are applied. The result of the letter bonuses and the individual tile scores are then summed. Lastly, any word bonuses are applied. It is also worth noting that if, in the case two Double Word bonuses are applied, the result of the summed word would be multiplied by four.

Other scoring rules:

- If a player uses all seven tiles in one placement an extra 50 points is awarded, in addition to the score of their placement.

- When the game ends, a final player's score is equal to their score, minus the sum of the tile scores they have remaining within their inventory of tiles.

- The individual tile scores of other words that may have been formed during a placement, are also included in a player's score.

Development – Tile scores

The first step towards calculating a player's score is clearly the definitions of score values for individual tiles, the most appropriate implementation for this can be achieved using a dictionary, similarly with the valid English word dictionary. Using each tile (alphabetic characters and a blank) as the unique key, and using its associated score as its value. Alongside this, I will produce a helper method that will interface with the dictionary, returning the score associated with the tile that is given.

Code

```
tileScores = {" ":0,"A":1,"B":3,"C":3,"D":2,"E":1,"F":4,"G":2,"H":4,
              "I":1,"J":8,"K":5,"L":1,"M":3,"N":1,"O":1,"P":3,"Q":10,
              "R":1,"S":1,"T":1,"U":1,"V":4,"W":4,"X":8,"Y":4,"Z":10}

def GrabTileScore(tile):
    #Theoretical error prevention
    if (not tile.upper() in tileScores):
        return 1
    return tileScores[tile.upper()]

print(GrabTileScore("q"))
```

Output

```
10
```

Code breakdown

Using official scores from the original game, storing them in `key:value` pairs. The function is a very simple addition that will be interfaced with, instead of using the dictionary directly. Just for the sake of safety, I've included a condition that should theoretically never be met, due to the validation that occurs before this function will be used. However, in the case that it does occur, the score used will simply be taken as one.

Development – Word scoring

This next step requires the incorporation of bonuses, tile scores, and all the formed words during a placement. Luckily, the calculations for the scores of additional formed words is simply, the sum of their individual tile scores. This is because, due to the fact that a bonus tile can only be used once, it is only applied to the main word that it is used within, and not to other words that branch off it, in other words, only the main word is affected by bonuses.

Firstly, to determine a word score. I developed a short function that takes the word as an argument, then for each letter within the word given, retrieve its associate score, storing it in an array. This array will be directly indexed to the word passed, for example, the word "HELLO", would return [4,1,1,1,1], where each element is at the same index position of its associate value.

Code

```
def DetermineLetterScores(word_in):
    #Storage of letter scores in-order
    letterScores = []
    for letter in word_in.upper():
        #Store associated score value
        letterScores.append(GrabTileScore(letter))
    return letterScores
print(DetermineLetterScores("HELLO"))
```

Output

```
[4, 1, 1, 1, 1]
```

Next, I required a method of distinguishing the tiles that were placed by the player, and those that already existed on the board (in the direction of the placement), as well as scoring them. The solution to this problem involved comparing the fully constructed word, to the player's inputted word, removing the duplicate occurrences from the fully constructed word. In a scenario that I may require the correct positioning of each word, this solution would not work. However, in this case, where only the number of occurrences of each tile is important (as I require the sum of the tile scores). I can instead remove the first/only occurrence of each shared letter.

Code

```
def GrabExistingTileScores(grid_in,userword_in,rot_in,coord_in):
    #Gather the before,within and after tiles, for this placement
    fullWord = DirectionalGather(grid_in,userword_in,rot_in,coord_in).upper()
    #For each letter in full word
    for x in fullWord:
        #If the same letter is used in the users input
        if x in userword_in:
            #Remove one occurance of it, not all
            fullWord = fullWord.replace(x.upper(),"",1)
    #Return the associate scores of the exisiting tiles
    return DetermineLetterScores(fullWord)
```

Code breakdown

This method works by gathering the fully constructed word, using the `DirectionalGather` function, then by iterating through each letter within the full word, it can then detect if it is shared with the user inputted word. If the letter is shared, it is removed/replaced with nothing, using the `replace` function attribute. By default, this function replaces all occurrences of the first parameter given with the second, but for this task, only one occurrence should be removed, so the optional argument `1`, is given to only remove one occurrence. Finally, using the function developed above, to return the summed tile scores of the existing tiles used during the placement (within the main placement).

Continuing with the next required method. I need to determine the tile scores of the words that may also be formed during a placement. The most suitable solution would be to firstly use the `GrabAllFormedWords` function, that I produced for the validation methods, which returns a list, containing the fully formed word, as well as any other words that may have been formed during the placement. I can then loop through the list of formed words that are returned by the function, with the exclusion of the first element, as it contains the fully formed word, which is scored separately.

Code

```
def CalculateSurroundingTileScore(grid_in,word_in,rot_in,indexedCoord):
    #Gather all surrounding words
    surroundingWords = GrabAllFormedWords(grid_in,word_in,rot_in,indexedCoord)
    #Remove the fullword (first element)
    del surroundingWords[0]
    scoreValue = 0
    #For each letter in word, append score
    for word in surroundingWords:
        for letter in word:
            scoreValue+=GrabTileScore(letter.upper())
    return scoreValue
```

Code breakdown

Firstly, storing a list of all the surrounding formed words and removing the first element, this can be safely done, as it is certain that each placement will contain at-least one letter, which is equal to this first element.

Then looping through each formed word (if any), then for each letter, within each word, sum its tile score. Returning the total score for surrounding words at the end.

The last prerequisite required now, is the adaptation of the `DetectBonusUsage` method, so that instead of displaying a debug message, it returns the bonuses that apply to this placement. A consideration that must be made, is that this subroutine is only used to detect the usage of bonuses, not to calculate the result of the bonus, this would involve returning the bonus type identifiers, such as: 1,3,2 and 6. However, in the case of letter bonuses, the score is dependent on the tile that landed on it. Meaning that, this function must return the bonus applied, and also the letter that landed on it.

Luckily, Python lists are not fixed to the data type they are declared as, making it very simple to (in this case), store an integer (bonus identifier), alongside a string/char (tile that landed on the bonus) within the same list. Despite this being a fairly poor programming practice, making the program slightly harder to follow, it makes the return handling much simpler, which is why I will use this technique.

The method I have designed, involves storing the bonuses used within a list, with the next element being the character that landed on said bonus. In other words, if a bonus is detected it will be stored at index position 0, and the tile associated with the bonus will be found by incrementing by one.

Example

`bonusesUsed = [bonusID,tileIndex,bonusID,tileIndex,bonusID,tileIndex]`

Adapted code

```
def DetectBonusUsage(bonusgrid_in,word_in,rot_in,coord_in):
    #Store each component of the coord
    rowIndex = coord_in[1]
    colIndex = coord_in[0]
    #Stores the bonus and the tile that used it
    bonusesUsed = []
    #Iterate for each letter within input
    for x in range(len(word_in)):
        #Detect and remove centre marker
        if (bonusgrid_in[rowIndex][colIndex] == "0"):
            bonusgrid_in[rowIndex][colIndex] = " "
        #Detect and remove bonus type
        if (bonusgrid_in[rowIndex][colIndex] != " "):
            #Store bonus type and index of tile
            bonusesUsed.append(int(bonusgrid_in[rowIndex][colIndex]))
            bonusesUsed.append(x)
            bonusgrid_in[rowIndex][colIndex] = " "
        #Increment correct index
        if (rot_in): #Right
            colIndex+=1
        else: #Down
            rowIndex+=1
    return bonusesUsed
```

Code breakdown

When a bonus is detected, the bonus character is converted to an integer, as it is stored within `bonusgrid_in` as a string, then stored within a list.

I realised the index position of the tile within the `word_in` would be more useful than the character itself. So, the index position of the tile that landed on the bonus is then also appended to the list.

Lastly, this list then returned. Allowing the bonuses to be evaluated and calculated in a further function.

Breakdown

Instead of displaying the outputs of this function, and the others above. Despite them being fairly clear anyway. I will instead wait until the scoring functions are combined, to calculate the final score for a placement, where I will show the result of each method.

An additional problem that I must also adapt for, is that there is currently no method of tracking which tiles (if any) entered by the user are blanks. This is relevant because when scoring a word, a blank has a score of 0, so a player would be scored for a tile, when they should not. If I were to continue development without solving this issue, blanks would be treated as regular tiles (with score values), and scoring would be significantly impacted due to inconsistency.

An elegant solution to this issue would be to mark the usage of blank tiles with another special character within the bonus grid or another grid altogether. However, due to limited time for this project, the simplest (and not very elegant) solution to this problem, is to apply the same usage of Python's unfixed type lists to the `DoesPlayerOwnTiles` function. Currently, this function outputs a debug message for the usage of a blank tile, as well as the tile that it is imitating. Simply by storing a list of the imitating tiles (if any), and returning the list, alongside the Boolean which is part of its primary task. Both of these variables would be encapsulated within a two-element list (theoretically an array) and returned.

Old code

```
def DoesPlayerOwnTiles(word_in,inventory_in):
    #Create a true copy of inventory
    tempInv = inventory_in.copy()
    #For each letter in word
    for x in word_in.upper():
        #If not using a skip character
        if (x != " "):
            #If player has tile
            if (x in tempInv):
                #Remove first occurrence of tile
                del tempInv[tempInv.index(x)]
            #Player doesn't have tile
            else:
                #If player has a blank
                if (" " in tempInv):
                    #Remove first occurrence of blank
                    print("Used blank for ",x)
                    del tempInv[tempInv.index(" ")]
                #Cannot make placement
                else:
                    print("You do not have ",x, "  ...
                    return False
    return True
```

Adapted code

```
def DoesPlayerOwnTiles(word_in,inventory_in):
    #Create a true copy of inventory
    tempInv = inventory_in.copy()
    #Stores letters used as blanks
    blankLetters = []
    #For each letter in word
    for x in word_in.upper():
        #If not using a skip character
        if (x != " "):
            #If player has tile
            if (x in tempInv):
                #Remove first occurrence of tile
                del tempInv[tempInv.index(x)]
            #Player doesn't have tile
            else:
                #If player has a blank
                if (" " in tempInv):
                    #Remove first occurrence of blank
                    print("Used blank for ",x)
                    #Store tile used as blank
                    blankLetters.append(x)
                    del tempInv[tempInv.index(" ")]
                #Cannot make placement
                else:
                    print("You do not have ",x, " to form this word")
                    return [False]
    return [True,blankLetters]
```

```
def ProcessTurn(playerletters_in, gamegrid_in, roundcounter_in, bonusgrid_in):
    ...
        tileData = DoesPlayerOwnTiles(userInputs.word,playerletters_in)
        #If word is constructable using their tiles
        if (tileData[0]):
            #If word doesn't overwrite another
            if (not DoesPlacementOverwrite(gamegrid_in,userInputs.word,userInputs.rotation,userInputs.coord)):
                ...
```

Code breakdown

As much as the solution is less than ideal, it will make the considerations for blanks much simpler. To reiterate, it works by storing the imitating tile used within the placement in a list named `blankLetters`, which is included within the successful return. There is no reason to include it within the failed return, as the `blankLetters` variable will never be used, unless if the player does own all the tiles required for the placement (as well as the other validation checks that occur before scoring).

The slight change of the `ProcessTurn` was also required to adapt to this different return, storing the return within a variable called `tileData`, where the first index contains the primary result of the function, and the second, containing blank data.

Development – Calculating the score

With all the required functions developed, as well as the required data included, I can now develop the function that combines them, returning the player's gained score at the end of a turn. It will involve using the same calculation that was detailed and analysed at the beginning of the Scoring sub system, in this order of operation:

Firstly, determine if blank tiles were used during the placement. If so, using the `tileData` variable, remove the occurrence of each tile that is imitated by a blank, using is index, confirming that they will not be scored. Next will be the usage of the `DetermineLetterScores` and the `GrabExistingTileScore` function, which will create lists, containing the score values of each letter within the users input, as well as the tiles that may occur before, within or after the main placement. The usage of bonuses can also be determined at this point using the `DetectBonusUsage` function.

Depending on if bonuses are used, the scoring of letters can be made. This will involve iterating through the `bonusID,tileIndex` pairs from the `DetectBonusUsage`, evaluating each for the different bonus types. So, in the scenario that a Double Letter bonus is found, the associated score for the tile at that index found be multiplied by two. Word bonuses are slightly different as they must be applied after the Letter bonuses, to ensure this, I can detect their usage in the same way, storing the number of occurrences of each type of word bonuses, then applying the bonus afterward. I could also perform the check for a player's usage of all seven tiles, where they will be awarded an extra fifty points.

Code

```
def CalculateScore(grid_in,bonusgrid_in,userword_in,tiledata_in,rot_in,coord_in):
    originalUserInput = userword_in
    #Replaces imitating letters with blanks
    #If blank tiles are used
    if (len(tiledata_in[1]) > 0):
        #For each blank used
        for blankUsed in tiledata_in[1]:
            #Replace this imitating tile with a blank tile character
            blankedUserInput = userword_in.replace(blankUsed," ",1)
        #Gather scores for only the user's input, accounting for blanks
        userInputTileScores = DetermineLetterScores(blankedUserInput)
    else: #Gather scores for only the user's input, without blanks
        userInputTileScores = DetermineLetterScores(originalUserInput)

    #Grab bonus type and the tile used pairs
    usedBonuses = DetectBonusUsage(bonusgrid_in,originalUserInput,rot_in,coord_in)
    #Gather scores of the existing tiles used (in main word)
    existingTileScores = GrabExistingTileScores(grid_in,originalUserInput,rot_in,coord_in)

    summedScore = 0 #Stores final score
    #Keeps track of number of word bonuses
    numDouble = 0
    numTriple = 0
    #For each item in bonuses
    index = 0
    #Applies and interprets bonuses
    #Only loops if bonuses are used
    while index < len(usedBonuses):
        if (usedBonuses[index] == 1): #Double Letter
            #Multiply this specific tile score by 2
            userInputTileScores[usedBonuses[index+1]]*=2
            print("Double letter")
        elif (usedBonuses[index] == 3): #Triple Letter
            userInputTileScores[usedBonuses[index+1]]*=3
            print("Triple letter")
        elif (usedBonuses[index] == 2): #Double Word
            numDouble+=1
            print("Double word")
        elif (usedBonuses[index] == 6): #Triple Word
            numTriple+=1
            print("Triple word")
        index+=2 #As they are stored in pairs
    #Sum tile scores
    for tileScore in userInputTileScores + existingTileScores:
        summedScore+=tileScore
    #Perform word bonuses, if used
    for x in range(numDouble):
        summedScore*=2
    for x in range(numTriple):
        summedScore*=3
    #If player uses all their tiles in one placement
    if (len(originalUserInput.replace(" ","")) == 7):
        summedScore+=50
        print("Used all tiles. Bonus 50 points!")
    #Gather potential surrounding word scores and return total
    surrScore = CalculateSurroundingTileScore(grid_in,originalUserInput,rot_in,coord_in)
    print("User's input: ",userInputTileScores)
    print("Existing: ",existingTileScores)
    print("Full word score:", summedScore)
    print("Surrounding:", surrScore)
    print("Total", summedScore+surrScore)
```

Replace imitating tiles (that may have been entered by the user) with blank characters. To prevent them from being scored as-if they were actual tiles. Then grab the base tile scores.

Grab the potential usage of bonuses within the user's input, storing the bonus type used, and the tile index as pairs.

Calculate and store the total score for the existing tiles that appear before, within and after the user's input. This is done separately, as bonuses will not occur in-place of existing tiles.

Then to account for the different operation required for each type of bonus, that may be encountered, the function iterates through each pair within the `usedBonuses` list, checking each bonus case. In the case that a Letter bonus is found, multiply the base tile score of the tile, at the index that the bonus is used. Using the second element of the pair, which equates to 2(n)+1. Word bonuses are slightly different, as they must be applied after all the Letter bonuses have been applied, so I have implemented a counter for the number of each type of bonus found. Which is applied later.

Next, the function loops through each tile score value for the user inputted word and existing tiles found (the fully formed word), by using list concatenation, adding the tile scores to the total.

At this point, the Word bonuses are applied. In the case that multiple word bonuses apply e.g. two Double Word bonuses, the score is equal to 4x the base score. Also, if the user enters seven tiles, they are awarded an extra 50 points.

Lastly, the base tile scores of the extra words, that may have been formed during the placement is determined. This is also calculated separately, as bonuses only apply to the main word, meaning that only the base tile scores are required. Finally, the surrounding tile scores, `surrScore`, and the `summedScore`, which contains the score of the fully formed word, including bonuses, is summed. Equalling the score earned during this placement.

Debug messages.

```
def ProcessTurn(playerletters_in, gamegrid_in, roundcounter_in, bonusgrid_in):
    ...
    #Process the valid placement
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(userInputs.coord)
    #Write to main grid
    PlaceWord(userInputs.rotation,userInputs.word,indexedCoord,gamegrid_in)
    #Calculate the score earned during this placement
    CalculateScore(gamegrid_in,bonusgrid_in,userInputs.word,tileData,userInputs.rotation,indexedCoord)
    #Remove used tiles
    LetterDeductor(playerletters_in,userInputs.word)
    #Re-distribute tiles
    Dealer(playerletters_in)
    return True
```

Not forgetting to add the call to the `ProcessTurn` method.

Outputs

```
It's Luc's turn
['W', 'I', 'O', 'N', 'G', 'P', 'D']
Enter the word to place
or enter "pass!" to forfeit your turn pond
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn h8
User's input:  [3, 1, 1, 2]
Existing:  []
Full word score: 7
Surrounding: 0
Total 7
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + P + + + 1 + + 6
 9 + + 1 + + + 1 O 1 + + + 1 + +
10 + 3 + + + 3 + N + 3 + + + 3 +
11 + + + + 2 + + D + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

```
It's Will's turn
['E', 'L', 'B', 'U', 'I', 'T', 'Z']
Enter the word to place
or enter "pass!" to forfeit your turn buil
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn d11
Double word
User's input:  [3, 1, 1, 1]
Existing:  [2]
Full word score: 16
Surrounding: 0
Total 16
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + P + + + 1 + + 6
 9 + + 1 + + + 1 O 1 + + + 1 + +
10 + 3 + + + 3 + N + 3 + + + 3 +
11 + + + B U I L D + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

```
It's Luc's turn
['W', 'I', 'G', 'R', 'O', ' ', 'E']
Enter the word to place
or enter "pass!" to forfeit your turn ers
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn i11
Used blank for  S
Double word
User's input:  [1, 1, 0]
Existing:  [3, 1, 1, 1, 2]
Full word score: 20
Surrounding: 0
Total 20
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + P + + + 1 + + 6
 9 + + 1 + + + 1 O 1 + + + 1 + +
10 + 3 + + + 3 + N + 3 + + + 3 +
11 + + + B U I L D E R S + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

```
It's Will's turn
['E', 'T', 'Z', 'R', 'I', 'E', 'T']
Enter the word to place
or enter "pass!" to forfeit your turn pass!
Forfeiting turn
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + P + + + 1 + + 6
 9 + + 1 + + + 1 O 1 + + + 1 + +
10 + 3 + + + 3 + N + 3 + + + 3 +
11 + + + B U I L D E R S + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

```
It's Luc's turn
['W', 'I', 'G', 'O', 'V', 'O', 'R']
Enter the word to place
or enter "pass!" to forfeit your turn o
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn i10
User's input:  [1]
Existing:  [1]
Full word score: 2
Surrounding: 2
Total 4
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + P + + + 1 + + 6
 9 + + 1 + + + 1 O 1 + + + 1 + +
10 + 3 + + + 3 + N O 3 + + + 3 +
11 + + + B U I L D E R S + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

```
It's Will's turn
['E', 'T', 'Z', 'R', 'I', 'E', 'T']
Enter the word to place
or enter "pass!" to forfeit your turn r t
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn g9
Double letter
Double letter
User's input:  [2, 0, 2]
Existing:  [1]
Full word score: 5
Surrounding: 3
Total 8
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + P + + + 1 + + 6
 9 + + 1 + + + R O T + + + 1 + +
10 + 3 + + + 3 + N O 3 + + + 3 +
11 + + + B U I L D E R S + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

Reference base tile scores (and distributions)

- 2 blank tiles (scoring 0 points)
- *1 point*: **E** ×12, **A** ×9, **I** ×9, **O** ×8, **N** ×6, **R** ×6, **T** ×6, **L** ×4, **S** ×4, **U** ×4
- *2 points*: **D** ×4, **G** ×3
- *3 points*: **B** ×2, **C** ×2, **M** ×2, **P** ×2
- *4 points*: **F** ×2, **H** ×2, **V** ×2, **W** ×2, **Y** ×2
- *5 points*: **K** ×1
- *8 points*: **J** ×1, **X** ×1
- *10 points*: **Q** ×1, **Z** ×1

Wikipedia

Sub system breakdown

Following through the outputs, I tested as many different cases as I could, given the random distribution of tiles. To begin with, placing the word "POND", which does not use any bonuses, only passing through the centre tile (outlined in red), so its score is equal to the sum of its individual tile scores, which is equal to: 3 + 1 + 1 + 2 = 7, which is correctly shown as the final score. It is worth noting that function of each debug message, so:

-User's input: Individual tile scores of the characters entered by the user, with Letter bonuses applied (if any).
-Existing: Individual tile scores of existing characters found: before, within or after, the user input.
-Full word score: The total score of the full main word, after all bonuses have been applied.
-Surrounding: The total score of all the potentially formed words that are formed during the placement.
-Total: The combined score of full word score and surrounding, equal to the final score.

Also, the usage of red outlines is used to highlight the location of a bonus tile that was used during the placement.

Continuing with the breakdown of the outputs. The second output shows the formation of the word "BUILD", by placing "BUIL", shown within the User's input output to have individual score values of: 3,1,1,1 and using the existing tile "D", which is shown to have a score value of 2, which is also correct. This brings the word score to 8, however, a Double word bonus is used at the letter "U", causing the original total to be doubled, the final score now equalling 16.

The next turn shows the usage of both a blank and a bonus, showing a similar process as the last, with the detection and scoring of existing tiles, noting that the blank tiles associate score within User's input is equal to 0, which is the intended, following the original Scrabble game.

Simulating a likely situation, where a user could not find a word to place, I decided to test the forfeit turn functionality, confirming that it still performs its task.

I then placed the letter "O" at I10, to form the main word "NO", as-well-as an additional formed word "OE", shown to have a score value of 2, which is summed with the Full word score, despite the fact that I do not think it is a valid English word, it is within the dictionary given and considered valid. Other than this fault with the dictionary, the surrounding word scoring method is shown to work.

The final output shows the usage of two bonuses, as-well-as the formation of an additional word. The formation of the full word "ROT", which is firstly scored as 1,0,1 containing a 0 because the user entered a skip character in-place of the "O", with the Existing debug message shows the score of the "O" as 1. The letters "R" and "T", both landed on a Double Letter bonus, causing the tile scores to become 2,0,2, giving a full word score of (2+1+2) 5. The surrounding word scoring then finds the score of the additional word "TOE", with a score value of 3. Finally, returning 8 as the total score.

Limitations in scoring and other issues

Clearly there are a few issues that arise with this method of scoring placements throughout the game. The first being the dictionary issue, seen in the output above, where "OE" was verified as a valid English word. I imagine there are many entries within the dictionary that are similar to this, but, also in the opposite case, whereby a valid English word is not listed within the dictionary. A solution to this problem could be the incorporation of user verification, allowing a user to decline a placement if they consider a word to be false, also allowing a user to accept a placement that would otherwise be marked as invalid by the program. However, incorporating this is out of the initial scope of the project, and would require extra time to design, develop and implement.

A logical issue that I have not experienced yet, during testing. However, is possible. Is that once a blank tile is placed, it is initially scored correctly for the player that made the placement. But, once it is placed on the board there is no method of detecting the difference between a blank tile and a true character tile. So, in the scenario that a user uses an existing tile, that was originally a blank. It would be treated and scored as-if it was a regular tile.

As much as this appears to be a significant flaw in the scoring, the chance of it occurring a quite low. Due to the fact that there are only two blank tiles distributed throughout an entire game, and that a user must then branch a word from it, for any effect to be seen. Because of this low frequency and the fairly minute impact this will have on the game, I will not allocate time to resolve the issue.

Stakeholder feedback

With the situation of the incorrect/absent entries within the dictionary, I do not think that it would be worthwhile to either: search for a replacement dictionary or modify entries within the current dictionary, due to its large scale. In addition, I feel that this is not a significant issue, as the large majority of words that will be placed will be accounted for within the dictionary, and that these minor cases will mostly be avoided by the user naturally. Although, I like the idea of flagging placements, as they are made, by the other players of the game, as it would include some verification from the user, I agree that it is not worth the development time.

I will have to agree with you, for this specific case, for the scoring of blanks. Similarly, to the dictionary issue, I am confident that this has a minor chance of occurring, and even when it does, its effect is minimal on the outcome of the game. So, again I do not think that this is worth the adjustment of scope.

Development – Tracking scores

Currently, the scoring method is fairly useless at the moment. Only displaying debug messages to the user, where the program requires the tracking of scores throughout the game, as-well-as determining the high scoring player at the end of the game. To achieve this, I will create another list, similarly to list containing player names. Which will be used to track/update individual player scores.

Code

```
def ProcessTurn(playerletters_in, gamegrid_in, roundcounter_in, bonusgrid_in,playerscore_in):
    ...
    #Calculate the score earned during this placement
    earnedScore = CalculateScore(gamegrid_in,bonusgrid_in,userInputs.word,
                                 tileData,userInputs.rotation,indexedCoord)
    #Remove used tiles
    LetterDeductor(playerletters_in,userInputs.word)
    #Re-distribute tiles
    Dealer(playerletters_in)
    return [True,earnedScore]

#Main program
...
#Create an entry for each player
playerScores = [0]*numPlayers
...
    #Call factory manager method
    turnOutcome = ProcessTurn(playerInventories[currentTurn],
                              gameGrid,roundCounter,bonusGrid,
                              playerScores[currentTurn])
    #If placement occurs increment round and modify score
    if (turnOutcome[0]):
        #Tracks number of successful placements
        roundCounter += 1
        #Increment score value
        playerScores[currentTurn] += turnOutcome[1]
    print("Score: ",playerScores[currentTurn])
...
```

Code breakdown

To keep track of each player's score, I have created a list of zeros at initialisation, for each player. At the end of each successful placement, the associate players score will be incremented by the amount gained.

Using a similarly poor implementation as with blank detection, by returning the `earnedScore` value as the second element of a list. Adjusting the main program to account for this change, it is definitely not ideal, and is equally a bad programming practice. However, it is coming to the end of the project, and so using these shortcuts is necessary for the completion of this project.

Outputs

```
It's Luc's turn
['O', 'T', 'I', 'U', 'L', 'M', 'T']
Enter the word to place
or enter "pass!" to forfeit your turn limo
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn h8
Score:  6
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + L + + + 1 + + 6
 9 + + 1 + + + 1 I 1 + + + 1 + +
10 + 3 + + + 3 + M + 3 + + + 3 +
11 + + + + 2 + + O + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

```
It's Luc's turn
['I', 'U', 'T', 'E', 'N', 'O', 'R']
Enter the word to place
or enter "pass!" to forfeit your turn ute
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn i8
Score:  10
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + L U T E 1 + + 6
 9 + + 1 + + + 1 I 1 + + + 1 + +
10 + 3 + + + 3 + M + 3 + + + 3 +
11 + + + + 2 + + O + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

Breakdown

To simplify the outputs, I have shown the result of the program for a single player. Firstly, scoring the player, 6 for the placement of "LIMO", which is correct. Then appending the score for the second word formation of "LUTE", which is 4. Showing, correctly that the players score is being increased by the expected amount (6+4 =10).

With the equivalent operation occurring for multiple players.

Sub system – Win condition

It is now time for the final system to be designed and implemented. Following the design brief and required operation. Its purpose will be to determine when any of the win conditions are met, being:

-A player using all of their available tiles, with none left to distribute.
-Each player has forfeited their turn, one after another.

Development – Running out of tiles

Starting with the first and most important win condition. Where there are no tiles remaining within the "bag" that can be distributed, and the player has successfully used all the tiles within their inventory. The implementation of this method would simply involve a condition within the main game loop, checking the size of the current player's inventory, after their turn has been made. If the inventory has a number of

elements, continue. Otherwise, if the inventory is empty, the main game loop condition will be switched, and then the ranking of player scores can take place, and the game will end.

Code

```
#Game Loop
while True:
    #Return back to first player index
    if (currentTurn % numPlayers == 0):
        currentTurn = 0
    #Display user data based on current turn
    print("It's " + playerNames[currentTurn] + "'s turn")
    #Display current users inventory
    print(playerInventories[currentTurn])
    #Call factory manager method
    turnOutcome = ProcessTurn(playerInventories[currentTurn],gameGrid...
    #If placement occurs increment round and modify score
    if (turnOutcome[0]):
        #Tracks number of successful placements
        roundCounter += 1
        #Increment score value
        playerScores[currentTurn] += turnOutcome[1]
    #Draw new state of the grid
    DrawGrid(gameGrid,bonusGrid)
    #If player has used all their tiles, end the game
    if (len(playerInventories[currentTurn]) == 0):
        print(playerNames[currentTurn]+" has used all their tiles!")
        break
    #Waits for input to slow down pace of game
    input("Tap enter")
    #Next player index
    currentTurn+=1
print("Game over")
```

Output

```
It's Luc's turn
['I', 'N', 'W']
Enter the word to place
or enter "pass!" to forfeit your turn win
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn k5
Double word
No tiles left in bag to distribute
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + W + + + +
 6 + 3 + + + 3 + + + 3 I + + 3 +
 7 + + 1 + + + 1 + 1 + N + 1 + +
 8 6 + + 1 + + + S A F E 1 + + 6
 9 + + 1 + + + 1 + 1 + + + 1 + +
10 + 3 + + + 3 + + + 3 + + + 3 +
11 + + + + 2 + + + + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
Luc has used all their tiles!
Game over
>>>
```

Code breakdown

After adding the detection for the win condition, I did not like the switching of the original *nobodyHasWon* Boolean, which was used as the loop condition. Due to the fact, that it allowed the loop to continue its execution, until the loop condition is re-evaluated, before exiting. So, instead I replaced the loop condition with *True*, using the *break* statement to exit the loop as soon as the condition is met.

Development – Forfeiting the game

The second win/end game condition, is the exiting of the main game loop, when each successive player has forfeited their turn using the *pass!* command. In this case, where each player is unable to make a placement, the game will end and the winner(s) will be decided from their ending scores. However, if, in the scenario that three-out-of-the-four players forfeit their turn, but the fourth makes a placement. The method of tracking successive forfeits will be reset, allowing the game to continue.

The implementation of this will involve adding a counter, that is incremented whenever a player forfeits their turn, and reset back to zero, whenever a successful placement is made. If this counter reaches the number of players within the game, a condition similar to the first win condition will be triggered, causing the game to end.

Code

```
passCounter = 0

#Game Loop
while True:
    #Return back to first player index
    if (currentTurn % numPlayers == 0):
        currentTurn = 0
    #Display user data based on current turn
    print("It's " + playerNames[currentTurn] + "'s turn")
    #Display current users inventory
    print(playerInventories[currentTurn])
    #Call factory manager method
    turnOutcome = ProcessTurn(playerInventories[currentTurn],gameGrid, ...
    #If placement occurs increment round and modify score
    if (turnOutcome[0]):
        #Tracks number of successful placements
        roundCounter += 1
        #Increment score value
        playerScores[currentTurn] += turnOutcome[1]
        #Reset forfeit counter
        passCounter = 0
    else: #User forfeited turn
        passCounter += 1
    #Draw new state of the grid
    DrawGrid(gameGrid,bonusGrid)
    #If player has used all their tiles, end the game
    if (len(playerInventories[currentTurn]) == 0):
        print(playerNames[currentTurn]+" has used all their tiles!")
        break
    #If everybody has forfeit
    if (passCounter == numPlayers):
        print("Nobody can make a placement!")
        break
    #Waits for input to slow down pace of game
    input("Tap enter")
    #Next player index
    currentTurn+=1
print("Game over")
```

Code breakdown

Luckily my implementation of the *ProcessTurn* function, makes the detection of a forfeited turn very simple, as it only returns *False* when the player enters the *pass!* command. Similarly, resetting the counter after every successful placement.

Also, adding the additional condition, required to detect the successive forfeiting of turns, breaking out of the game loop.

Outputs

```
Enter the number of players 2
Enter player name Luc
Enter player name Will
...
It's Luc's turn
['N', 'O', 'E', 'D', 'T', 'Y', ' ']
Enter the word to place
or enter "pass!" to forfeit your turn pass!
Forfeiting turn
...
It's Will's turn
['S', 'Z', 'L', 'E', 'E', 'S', 'L']
Enter the word to place
or enter "pass!" to forfeit your turn pass!
Forfeiting turn
...
Nobody can make a placement!
Game over
>>>
```

passCounter = 1

passCounter = 1

passCounter = 0

passCounter = 2

passCounter == numPlayers

passCounter = 1

passCounter = 2

passCounter == numPlayers

```
Enter the number of players 2
Enter player name Luc
Enter player name Will
...
It's Luc's turn
['I', 'B', 'E', 'E', 'V', 'I', 'K']
Enter the word to place
or enter "pass!" to forfeit your turn pass!
Forfeiting turn
...
It's Will's turn
['T', 'I', 'O', 'U', 'I', 'I', 'P']
Enter the word to place
or enter "pass!" to forfeit your turn PIT
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn H8
...
It's Luc's turn
['I', 'B', 'E', 'E', 'V', 'I', 'K']
Enter the word to place
or enter "pass!" to forfeit your turn pass!
Forfeiting turn
...
It's Will's turn
['O', 'U', 'I', 'I', 'I', 'W', 'E']
Enter the word to place
or enter "pass!" to forfeit your turn pass!
Forfeiting turn
...
Nobody can make a placement!
Game over
>>>
```

Breakdown

With the win conditions implemented and working as expected, allowing the game to actually end. By utilising the *passCounter* variable to track successive forfeited turns, which will work for any number of players. In addition to the true win condition, detecting the usage of all of a player's tiles, before ending the game.

Development – Final scoring

The last feature to be implemented, before the project is complete, is the scoring and displaying of rankings at the end of the game. This will firstly involve the calculation of a player's final score. Which for the player that ended the game (by using all of their tiles), is simply equal to their score at the end of the game. But, for the other players, and in the case of a forfeited game, is equal to their score at the end of the game, minus the tile scores that they have remaining.

As well as this, I will produce a small method that determines the highest scoring player, and announces them as the winner. This can be implemented by looping through the players scores, and each time the highest value is exceeded, replace it with the new highest value.

Code

```
successfulEndType = False
#Game Loop
while True:
    ...
    #If player has used all their tiles, end the game
    if (len(playerInventories[currentTurn]) == 0):
        print(playerNames[currentTurn]+" has used all of their tiles!")
        successfulEndType=True
        break
    #If everybody has forfeit
    if (passCounter == numPlayers):
        print("Nobody can make a placement!")
        successfulEndType=False
        break
    ...

...
print("Game over")
#Adjust final scores depending on outcome
print(playerScores)
print(CalculateEndScores(successfulEndType,playerInventories,playerScores,playerNames,currentTurn))
```

```
def CalculateEndScores(issuccessfulend,playerinv_in,playerscores_in,playernames_in,endingturn_
    #For the number of players in the game
    for x in range(len(playernames_in)):
        #If game has a "winner" and looking at them
        if (issuccessfulend and x != endingturn_in):
            continue #Skip
        #Foreach tile in player's inv
        for tile in playerinv_in[x]:
            #Deduct each tiles score
            playerscores_in[x] -= GrabTileScore(tile)
    return playerscores_in
```

Output

```
It's Luc's turn
['W', 'T', 'I', 'F']
Enter the word to place
or enter "pass!" to forfeit your turn pass!
Forfeiting turn
...

It's Will's turn
['A']
Enter the word to place
or enter "pass!" to forfeit your turn a
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn i8
No tiles left in bag to distribute
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + W A + + 1 + + 6
 9 + + 1 + + + 1 I 1 + + + 1 + +
10 + 3 + + + 3 + S + 3 + + + 3 +
11 + + + + 2 + F E R N 2 + + + +
12 1 + + 2 + + A 1 + + + 2 + + 1
13 + + 2 + + + E + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
Will has used all of their tiles!
Game over
[7, 19]
[-3, 19]
```

[Luc, Will]

Original score values.

Updated score values.

Breakdown

The additions worth noting, include a Boolean (`successfulEndType`), which is used to track which state the game ended in, being: a forfeited end, or a player's usage of all their tiles. Using `False`, `True` respectively. This value is then passed along the various player data variables, as well as the index of the player that the game ended on, `currentTurn`. Into the newly developed `CalculateEndScores` function, which depending on the games end type, will either; deduct the remaining tile scores from each player's inventory, from their total score, or in the case that the game did not end by forfeit, the player that used all their tiles is not deducted from.

Reviewing the output, it can be seen that the game has a "winner", being Will. So, his final score is unchanged, remaining at nineteen. Whereas my (Luc's) score is deducted by the tile scores of my remaining tiles, which is equal to ten. [W,T,I,F] = [4,1,1,4] = 10.

Development – Determining the winner

Lastly, I need produce a short method that will compare each player's final score. Finding the highest scoring player and their score value, and outputting it. The process of this method, will involve performing a linear search throughout the length of the `playerScores` list, storing the index of the highest value found. Then outputting the score and player name at that index found, which will be announced as the winner.

Code

```
def DetermineWinner(playerscores_in,playernames_in):
    highestIndex = 0
    for index in range(len(playernames_in)):
        #If found score is higher
        if (playerscores_in[index] > playerscores_in[highestIndex]):
            highestIndex = index #Store index of highest score
    print(playernames_in[highestIndex], " is the winner!")
    print("with ",playerscores_in[highestIndex]," points!")
```

Output

```
It's Rob's turn
['F', 'N']
Enter the word to place
or enter "pass!" to forfeit your turn f n
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn k9
Double letter
No tiles left in bag to distribute
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + T R E E S + + 6
 9 + + 1 + + + 1 + 1 + F A N + +
10 + 3 + + + 3 + + + 3 + W + 3 +
11 + + + + 2 + + + + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
Rob has used all of their tiles!
Game over
[6, 18]
[-5, 18]
Rob  is the winner!
with  18  points!
```

[Luc,Rob]

Breakdown

By using the same method as above to quickly test the end game, achievable by changing the `availableTiles` variable to only contain [number of players' x 7] tiles. Which is also the reasoning for the negative end scores seen.

However, it can be seen that the method works, outputting the true winner of the game, as well as their final score. But, there are a few minor issues with this solution, as it is obviously possible that two players can share the same score, in which they should both be announced as the join winners. This solution will only be selecting the first player found with the highest score, which I think is adequate.

Sub system - Final improvements

At this point in development the game is technically finished, as all systems are in-place and the game works! However, there are a few minor quality-of-life improvements and minor bug fixes that I would like to implement, before ending development on the product.

Empty inputs

The first, is a fairly major issue encountered during testing, whereby the user can enter a word input, that only contains spaces, or even an input of nothing (technically "\n" as a new line/escape character). As seen below.

In the case of a newline character

```
It's Will's turn
['M', 'P', 'N', 'E', 'R', 'L', 'R']
Enter the word to place
or enter "pass!" to forfeit your turn
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn ill
Traceback (most recent call last):
  File "I:\Py\Scrabble\Documentation\Sub Systems\_WordPlacingBuilding.py", line
756, in <module>
    turnOutcome = ProcessTurn(playerInventories[currentTurn],gameGrid,roundCount
er,bonusGrid,playerScores[currentTurn])
  File "I:\Py\Scrabble\Documentation\Sub Systems\_WordPlacingBuilding.py", line
372, in ProcessTurn
    earnedScore = CalculateScore(gamegrid_in,bonusgrid_in,userInputs.word,tileDa
ta,userInputs.rotation,indexedCoord)
  File "I:\Py\Scrabble\Documentation\Sub Systems\_WordPlacingBuilding.py", line
458, in CalculateScore
    surrScore = CalculateSurroundingTileScore(grid_in,originalUserInput,rot_in,c
oord_in)
  File "I:\Py\Scrabble\Documentation\Sub Systems\_WordPlacingBuilding.py", line
467, in CalculateSurroundingTileScore
    del surroundingWords[0]
IndexError: list assignment index out of range
>>>
```

In the case of only spaces

```
It's Luc's turn
['I', 'G', 'L', 'S', 'E', 'S', 'I']
Enter the word to place
or enter "pass!" to forfeit your turn
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn ill
++++  is not a valid word
Enter the word to place
or enter "pass!" to forfeit your turn
```

In the scenario that a new line character is used/enter, this issue can cause the program to break, meaning that this is an issue that must be resolved. In the second instance, where only spaces are used, the program outputs the empty cell characters, which is also an unwanted outcome.

This can be detected an avoided within the `ValidateWordInput` function, by adding an extra condition, to check whether the input only contains only spaces, using the `isspace` method attribute. Luckily enough, this method also registers a new line character as a space. So, this solution will resolve the issue for both cases.

Code

```
def ValidateWordInput():
    #Repeat until a valid input is given
    while True:
        #Retrieve word from user in uppercase
        wordAttempt = input("Enter the word to place\nor enter \"pass!\" to forfeit your turn ").upper()
        #First check for forfeit command
        if (wordAttempt == "PASS!"):
            return Event.SKIPTURN
        #If string contains only spaces and alphabetic characters
        if (all(letter.isalpha() or letter.isspace() for letter in wordAttempt)):
            #If string doesnt only contain space/newline characters
            if (not all(letter.isspace() for letter in wordAttempt)):
                if (len(wordAttempt) <= 15): #If less than 15 chars
                    return wordAttempt #Return uppercase validword
                else:
                    print("The word entered MUST be less than 15 characters")
            else:
                print("The word MUST contain atleast one letter")
        else:
            print("The word MUST only contain letters and spaces")
```

Code breakdown

Again, utilising the inline loop, to run through each character within the input, gathering the result of the `isspace` method for each letter. Finally, determining if all the results gathered are `True`, using the `all` function, and using the binary opposite result. Performing, the necessary check. Outputting a meaningful error message, allowing the player to retry.

Output

```
It's Will's turn
['B', 'E', 'D', 'D', 'O', 'P', 'R']
Enter the word to place
or enter "pass!" to forfeit your turn
The word MUST contain atleast one letter
Enter the word to place
or enter "pass!" to forfeit your turn
The word MUST contain atleast one letter
Enter the word to place
or enter "pass!" to forfeit your turn po
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn fll
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + G + + + 1 + + 6
 9 + + 1 + + + 1 U 1 + + + 1 + +
10 + 3 + + + 3 + I + 3 + + + 3 +
11 + + + + 2 P O D + + 2 + + + +
12 1 + + 2 + + + E + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

Attempting to input only spaces, and a newline character.

Output display

My second point is more of a quality-of-life improvement. While running through the program's output, it became clear that the outputs require some more spacing, as currently it appears to take the form of a large block of text, which is hard to follow, especially for a new user. In addition to this, some of the debug messages need to be removed, and I would also like to make the output of a player's tiles clearer, firstly by formatting it, so that it is not a list.

Spacing

Current output

```
Dictionary build successful. Dictionary size: 370101
Enter the number of players, 1-42
Enter player name Luc
Enter player name Will
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + 0 + + + 1 + + 6
 9 + + 1 + + + 1 + 1 + + + 1 + +
10 + 3 + + + 3 + + + 3 + + + 3 +
11 + + + + 2 + + + + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + I + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
It's Luc's turn
['I', 'E', 'L', 'S', 'T', 'C', 'R']
Enter the word to place
or enter "pass!" to forfeit your turn
The word MUST contain atleast one letter
Enter the word to place
or enter "pass!" to forfeit your turn lets
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn g8
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + I + + + 1 + +
 8 6 + + 1 + + L E T S + 1 + + 6
 9 + + 1 + + + 1 + 1 + + + 1 + +
10 + 3 + + + 3 + + + 3 + + + 3 +
11 + + + + 2 + + + + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter
It's Will's turn
['T', 'P', 'E', 'X', 'T', 'O', 'N']
Enter the word to place
or enter "pass!" to forfeit your turn on
Enter "1" for right, "2" for down
or "RE" to reattempt your turn re
Reattempting inputs
Enter the word to place
or enter "pass!" to forfeit your turn p n
Enter "1" for right, "2" for down
or "RE" to reattempt your turn 2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn h7
   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + I + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + I + + + I P I + + + 1 + +
 8 6 + + 1 + + L E T S + 1 + + 6
 9 + + 1 + + + 1 N 1 + + + 1 + +
10 + 3 + + + 3 + + + 3 + + + 3 +
11 + + + + 2 + + + + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + I + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter
```

Improved output

```
The dictionary has 370101 words
Enter the number of players, 1-4 2
Enter player 1 name Luc
Enter player 2 name Will

   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + 0 + + + 1 + + 6
 9 + + 1 + + + 1 + 1 + + + 1 + +
10 + 3 + + + 3 + + + 3 + + + 3 +
11 + + + + 2 + + + + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
It's Luc's turn!
-------Your Letters-------
| R | E | P | A | O |   | U |
Enter the word to place
or enter "pass!" to forfeit your turn
harps
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
h8
>Used blank for S
>Double letter bonus!
>No tiles left in bag to distribute

   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + E + + + 1 + + 6
 9 + + 1 + + + 1 A 1 + + + 1 + +
10 + 3 + + + 3 + R + 3 + + + 3 +
11 + + + + 2 + + P + + 2 + + + +
12 1 + + 2 + + + S + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Will's turn!
-------Your Letters-------
| A | E | T | I | L | R | E |
Enter the word to place
or enter "pass!" to forfeit your turn
tiler
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
h8
!You cannot overwrite tiles!
Enter the word to place
or enter "pass!" to forfeit your turn
ailer
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
i12
>Double word bonus!
>No tiles left in bag to distribute

   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + H + + + 1 + + 6
 9 + + 1 + + + 1 A 1 + + + 1 + +
10 + 3 + + + 3 + R + 3 + + + 3 +
11 + + + + 2 + + P + + 2 + + + +
12 1 + + 2 + + + S A I L E R + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter
```

Luc Heaton

Breakdown

Comparing the two sets of outputs, it is clear that the adjustments make a large impact on the readability of the outputs. The changes that were made include:

-Removal/modification of debug messages.
-When entering player names, a player number is also displayed.
-Formatted tile inventory display.
-Optimised and reused code.
-Correct spacing during various inputs and outputs.
-Extra informative outputs are marked with a ">".
-Error messages are marked with surrounding exclamation marks.

I will not be displaying the small additions to the code that were made in-order to produce this clearer output, as the additions/modifications were all very short, and the methods used have already been documented and explained.

User guide

Although this is not within the initial scope of the project, I feel that the addition of some useful tips, such as a list of the tile scores, as well as some other useful information surrounding the method of play, would improve the user experience substantially.

```
The dictionary has 370101 words
Enter the number of players, 1-4 2
Enter player 1 name Luc
Enter player 2 name Will

   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + 0 + + + 1 + + 6
 9 + + 1 + + + 1 + 1 + + + 1 + +
10 + 3 + + + 3 + + + 3 + + + 3 +
11 + + + + 2 + + + + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
It's Luc's turn!
-------Your Letters-------
|   | O | T | R | E | H | I |
Enter the word to place
or enter "pass!" to forfeit your turn
```

Observing the current layout of the output, it can be noted that, there is a large amount of unused space beside the game grid. It is within this space, that extra information could be displayed. The main data that I would like to display, may include:

-The individual tile scores of each tile.
-A key for the different bonus types, as currently they are arbitrary.
-Some gameplay mechanics, such as, entering a space in place of an existing character.

As I would like to display this data each time the grid is drawn, it would be suitable to append this system within the `DrawGrid` method. The technique that I have in-mind to incorporate this, involves storing all the text that will be displayed within a list. Then during the drawing of the grid, (as it is drawn progressively, line by line) the process will iterate through the list of text, displaying each element, on each successive line. Displaying the data to the user, and making use of the empty space.

Code

```
def DrawGrid(grid_in,bonus_in):
    print("") #Start new line
    #Store for sidebar text
    sidebarHelp = [" Letter Scoring:","[A]1  [B]3  [C]3  [D]2  [E]1  [F]4",
                   "[G]2  [H]4  [I]1  [J]8  [K]5  [L]1","[M]3  [N]1  [O]1  [P]3  [Q]10 [R]1",
                   "[S]1  [T]1  [U]1  [V]4  [W]4  [X]8","[Y]4  [Z]10 [ ]0","Bonus types:",
                   "TWS:6  DWS:2  TLS:3  DLS:1","Helpful Tips:",
                   "1. The coordinate is the first letter's", "position.",
                   "2. When using a placed letter, use a [space]","in its place.",
                   "3. When using a blank, enter the letter","as if you have it."]
    #Output top row coords
    print("   A B C D E F G H I J K L M N O", end = "")
    #For each item in board
    for y in range(15):
        if (y < len(sidebarHelp)): #If item in sidebarHelp
            #Print with space to the left
            print(" "*2+sidebarHelp[y],end="")
        for x in range(15):
            if (x % 15 == 0): #If at end of the row
                print("") #Start new line
                print("%2d" % (y+1,), end = " ")
            #If bonus contents is blank
            if (bonus_in[y][x] == " "):
                #Print game grid
                print(grid_in[x][y], end = " ")
            else: #If bonus contents is populated
                #Print special character
                print(bonus_in[y][x], end = " ")
    print("") #Start new line
```

Code breakdown

To begin with, the addition of a list, *sidebarHelp*, containing each string that will be outputted on each line. Paired with an extra condition within the outer loop, which executes for every row. Which detects if there is another element within the list, in which its content in printed, along with two spaces to the left of the string achieved by using, *" "*2*.

Output

```
The dictionary has 370101 words
Enter the number of players, 1-4 2
Enter player 1 name Luc
Enter player 2 name Will

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + + 0 + + + 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 + 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + 3 + + + 3 +   position.
11 + + + + 2 + + + + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
It's Luc's turn!
-------Your Letters-------
| E | I | I | W | F | P | L |
Enter the word to place
or enter "pass!" to forfeit your turn
|
```

Breakdown

I think that this was a-much-needed implementation, by helping the player understand the program. Especially the addition of the tile scores, and bonus types, as it allows players to make better decisions during placements. The "Helpful Tips", also felt like a necessary addition, clarifying the method of play to the user, without requiring a full tutorial.

Luc Heaton

Finished product

After the many smaller additions, improvements, and optimisations to the program. I am glad to say that the project has finished development, and the program is in its complete form. With all sub systems implemented, also, a few additional features, not in the project's initial scope.

Full code

```
##Modules##
import random
import copy
import ast
##Methods##
#Determine and output the highest scoring player as the winner
def DetermineWinner(playerscores_in,playernames_in):
    highestIndex = 0
    for index in range(len(playernames_in)):
        #If found score is higher
        if (playerscores_in[index] > playerscores_in[highestIndex]):
            highestIndex = index #Store index of highest score
    print(playernames_in[highestIndex], " is the winner!")
    print("with ",playerscores_in[highestIndex]," points!")

#Depending on the end type, calculate final scores
def CalculateEndScores(issuccessfulend,playerinv_in,playerscores_in,playernames_in,endingturn_in):
    #For the number of players in the game
    for x in range(len(playernames_in)):
        #If game has a "winner" and looking at them
        if (issuccessfulend and x == endingturn_in):
            continue #Dont try to deduct tile scores
        #Foreach tile in player's inv
        for tile in playerinv_in[x]:
            #Deduct each tiles score
            playerscores_in[x] -= GrabTileScore(tile)
    #Return updated player scores
    return playerscores_in

#Called after LetterDeductor, to re-distibute tiles
#for a sum of 7, if tiles are available
def Dealer(playerLetters_in):
    #If the bag is not empty
    if (len(availableTiles)>0):
        #Calc number to deal
        numToDeal = 7 - len(playerLetters_in)
        #If not enough in bag to fill up players collection
        #give all remaining
        if (numToDeal > len(availableTiles)):
            numToDeal = len(availableTiles)
        #Deal amount to player and remove from bag
        for x in range(numToDeal):
            #Pick random index within range
            randomIndex = random.randint(0,len(availableTiles)-1)
            #Add letters to local inventory variable
            playerLetters_in.append(availableTiles[randomIndex])
            #Remove from list
            del availableTiles[randomIndex]
    else: #if bag is empty
        print(">No tiles left in bag to redistribute")
    #Returns new inventory
    return playerLetters_in

#Takes a player's inventory, and the word successfuly placed
#Removing the tiles used to form the word from the inventory
def LetterDeductor(playerLetters_in, wordPlaced):
    #Convert to uppercase
    wordPlaced = wordPlaced.upper()
    #For each letter in the word
    for x in wordPlaced:
        #Ignore blank value
        if (x != " "):
            #Attempt to remove letter
            try:
                #Grab letters index (first instance)
                letterIndex = playerLetters_in.index(x)
                #Remove from player
                del playerLetters_in[letterIndex]
            #If letter doesn't exist, assume blank tile used
            except:
                #If can't remove letter, remove blank tile
                del playerLetters_in[playerLetters_in.index(" ")]
    #Doesnt require a return as the value is modified directly (pass by reference)
```

```
#Gather the score gained from extra formed words, during a placement
def CalculateSurroundingTileScore(grid_in,word_in,rot_in,indexedCoord):
    #Gather all surrounding words
    surroundingWords = GrabAllFormedWords(grid_in,word_in,rot_in,indexedCoord)
    #Remove the fullword (always at first element)
    del surroundingWords[0]
    scoreValue = 0
    #For each letter in found word, append its tile score
    for word in surroundingWords:
        for letter in word:
            scoreValue+=GrabTileScore(letter.upper())
    #Return total
    return scoreValue
#Returns a tiles associate score
def GrabTileScore(tile):
    #Theoretical error prevention
    #for a non-existent tile
    if (not tile.upper() in tileScores):
        return 1
    #Return associate tile score
    return tileScores[tile.upper()]

#Return an array of tile scores for the fully formed word
def GrabExistingTileScores(grid_in,userword_in,rot_in,coord_in):
    #Gather the before,within and after tiles, for this placement
    fullWord = DirectionalGather(grid_in,userword_in,rot_in,coord_in).upper()
    #For each letter in full word
    for x in fullWord:
        #If the same letter is used in the user's input
        if x in userword_in:
            #Remove one occurance of it, not all (third param)
            fullWord = fullWord.replace(x.upper(),"",1)
    #Return the associate scores of the exisiting tiles
    return DetermineLetterScores(fullWord)

#Determines the bonuses that have been used in this placement
#as well as the index of the tiles that used them
def DetectBonusUsage(bonusgrid_in,word_in,rot_in,coord_in):
    #Store each component of the coord
    rowIndex = coord_in[1]
    colIndex = coord_in[0]
    #Stores the bonus and the tile that used it
    bonusesUsed = []
    #Iterate for each letter within input
    for x in range(len(word_in)):
        #Detect and remove centre marker
        if (bonusgrid_in[rowIndex][colIndex] == "0"):
            bonusgrid_in[rowIndex][colIndex] = " "
        #Detect and remove bonus type
        if (bonusgrid_in[rowIndex][colIndex] != " "):
            #Store bonus type and index of tile
            bonusesUsed.append(int(bonusgrid_in[rowIndex][colIndex]))
            bonusesUsed.append(x)
            bonusgrid_in[rowIndex][colIndex] = " "
        #Increment correct index
        if (rot_in): #Right
            colIndex+=1
        else: #Down
            rowIndex+=1
    return bonusesUsed

#Given a string, calculate its base score values
def DetermineLetterScores(word_in):
    #Storage of letter scores in-order
    letterScores = []
    for letter in word_in.upper():
        #Store associated score value
        letterScores.append(GrabTileScore(letter))
    return letterScores
```

Luc Heaton

```
#Determine the score at the end of a successful placement
def CalculateScore(grid_in,bonusgrid_in,userword_in,tiledata_in,rot_in,coord_in):
    originalUserInput = userword_in
    #Replaces imitating letters with blanks
    #If blank tiles are used
    if (len(tiledata_in[1]) > 0):
        #For each blank used
        for blankUsed in tiledata_in[1]:
            #Replace this imitating tile with a blank tile character
            blankedUserInput = userword_in.replace(blankUsed," ",1)
        #Gather scores for only the user's input, accounting for blanks
        userInputTileScores = DetermineLetterScores(blankedUserInput)
    else: #Gather scores for only the user's input, without blanks
        userInputTileScores = DetermineLetterScores(originalUserInput)

    #Grab bonus type and the tile used, as pairs
    usedBonuses = DetectBonusUsage(bonusgrid_in,originalUserInput,rot_in,coord_in)
    #Gather scores of the existing tiles used (in main word)
    existingTileScores = GrabExistingTileScores(grid_in,originalUserInput,rot_in,coord_in)

    summedScore = 0 #Stores final score
    #Keeps track of number of word bonuses
    numDouble = 0
    numTriple = 0
    #For each item in bonuses
    index = 0
    #Applies and interprets bonuses
    #Loops if bonuses are used
    while index < len(usedBonuses):
        if (usedBonuses[index] == 1): #Double Letter
            #Multiply this specific tile score by 2
            userInputTileScores[usedBonuses[index+1]]*=2
            print(">Double letter bonus!")
        elif (usedBonuses[index] == 3): #Triple Letter
            #Multiply this specific tile score by 3
            userInputTileScores[usedBonuses[index+1]]*=3
            print(">Triple letter bonus!")
        elif (usedBonuses[index] == 2): #Double Word
            #Increment double word bonus counter
            numDouble+=1
            print(">Double word bonus!")
        elif (usedBonuses[index] == 6): #Triple Word
            #Increment triple word bonus counter
            numTriple+=1
            print(">Triple word bonus!")
        index+=2 #As they are stored in pairs
    #Combine tile scores
    for tileScore in userInputTileScores + existingTileScores:
        summedScore+=tileScore
    #Perform word bonuses, if used
    for x in range(numDouble):
        summedScore*=2
    for x in range(numTriple):
        summedScore*=3
    #If player uses all their tiles in one placement
    #Skip characters are remove, as they don't count
    if (len(originalUserInput.replace(" ","")) == 7):
        summedScore+=50
        print(">Used all tiles. Bonus 50 points!")
    #Gather potential surrounding word scores and return total base tile score
    surrScore = CalculateSurroundingTileScore(grid_in,originalUserInput,rot_in,coord_in)
    #Return total score
    return summedScore+surrScore

#Is string in dictionary?
def IsValueInDictionary(value):
    #Using global variable dictionary
    #Dictionary is in lowercase
    if (value.lower() in dictionary):
        return True
    return False

#Given a list of words check if all of the
#words are within the dictionary
def AreWordsInDictionary(words_in):
    #Check each words is a true word
    for word in words_in:
        #If word not in dictionary
        if (not IsValueInDictionary(word)):
            #As soon as one if not valid, none are
            print("!",word, " is not a valid word!")
            return False
    return True
```

```
#Gather any connecting tiles to a placement
#that occur before the placement
def BeforeWordTiles(section_in,indexcoord_in,rotation_in):
    #Determine index required
    if (rotation_in): #Right placement
        #Deduct one, to not include first character of placement
        indexPointer = indexcoord_in[0]-1
    else: #Down placement
        indexPointer = indexcoord_in[1]-1
    #If equal to -1/placement at border
    #return nothing found
    if (indexPointer < 0):
        return ""
    #Declare list of chars found
    charsBehind=[]
    #Loop if not outside boundary
    while (indexPointer >= 0):
        #If current cell is not empty
        if (section_in[indexPointer] != "+"):
            #Store value of current index in list
            charsBehind.append(section_in[indexPointer])
            #Decrement index
            indexPointer-=1
        else: #If hit empty cell
            break
    #Reverse the list and return as a string
    return ''.join(reversed(charsBehind))

#Gather any connecting tiles to a placement
#that occur within the placement, at skip characters
def WithinWordTiles(section_in,word_in,indexcoord_in,rotation_in):
    #Determine index required
    if (rotation_in): #Right placement
        #Starting from first character in placement
        indexPointer = indexcoord_in[0]
    else: #Down placement
        indexPointer = indexcoord_in[1]
    #Declare list of chars found
    fullWord=[]
    #Iterate for amount of chars within user input
    for x in range(len(word_in)):
        #Store element at index position (iterator+start index)
        fullWord.append(section_in[x+indexPointer])
    return ''.join(fullWord)

#Gather any connecting tiles to a placement
#that occur after the placement
def AfterWordTiles(section_in,word_in,indexcoord_in,rotation_in):
    #Determine index required
    if (rotation_in): #Right placement
        #Starting from 1+last character in placement
        indexPointer = indexcoord_in[0]+len(word_in)
    else: #Down placement
        indexPointer = indexcoord_in[1]+len(word_in)
    #If placement past border, return empty
    if (indexPointer > 14):
        return ""
    #Declare list of chars found
    charsAfter=[]
    #Loop if not past boundary
    while (indexPointer < 15):
        #If current cell is not empty
        if (section_in[indexPointer] != "+"):
            #Store value of current index in list
            charsAfter.append(section_in[indexPointer])
            #Increment index
            indexPointer+=1
        else: #If hit a empty cell
            break
    #Return as a string
    return ''.join(charsAfter)

#Given a grid to slice, the orientation of the slice and starting position
#Return the specified slice of the grid given
def GrabSection(gridToRead, orientation_in, startPos_in):
    tempSection=[] #Stores section as it is being read
    #If right grab row
    if (orientation_in):
        for x in range(15):
            tempSection.append(gridToRead[x][startPos_in[1]])
    #If down grab column
    else:
        for x in range(15):
            tempSection.append(gridToRead[startPos_in[0]][x])
    return tempSection
```

```
#Search in placement direction for tiles
#Return fully constructed word
def DirectionalGather(grid_in,word_in,rotation_in,coord_in):
    #Grab the section that th eplacement is within
    placedSection = GrabSection(grid_in,rotation_in,coord_in)
    #Gather data from each direction
    beforeTiles = BeforeWordTiles(placedSection,coord_in,rotation_in)
    withinTiles = WithinWordTiles(placedSection,word_in,coord_in,rotation_in)
    afterTiles = AfterWordTiles(placedSection,word_in,coord_in,rotation_in)
    return beforeTiles+withinTiles+afterTiles

#Performs checks in placement direction and surrounding
#to gather all the formed words, during a placement
def GrabAllFormedWords(grid_in,word_in, rotation_in,startpos_in):
    #Store each word formed
    formedWords = []
    #Store and grab main word
    formedWords.append(DirectionalGather(grid_in,word_in,rotation_in,startpos_in))
    #Create copy of the original startPos
    iteratingStartPos = startpos_in.copy()
    #Get the opposite of placement rotation
    if (rotation_in): #Right
        oppositeRot = False
    else:
        oppositeRot = True
    #For length of players input
    for letter in word_in.upper():
        #If player used a skip character, ignore
        #and iterate to next letter
        if (letter != " "):
            #Gather potential tiles at current letter
            wordFound = DirectionalGather(grid_in,letter,oppositeRot,iteratingStartPos)
            #Store result
            formedWords.append(wordFound)
        #Increment in opposite direction to placement
        if (oppositeRot):#If original is down
            iteratingStartPos[1]+=1
        else: #If original is right
            iteratingStartPos[0]+=1
    #Filter single character words
    #returning only relevant words
    filteredWords = []
    for entry in formedWords:
        #If entry is more than one character in length
        if (len(entry) > 1):
            filteredWords.append(entry)
    return filteredWords

#Orientation: True=right False=down
#Assign a word to its specified position within a given 2D array
def PlaceWord(orientation,word_in,startPos, grid_in):
    startPosTemp = startPos.copy()
    #Temp capitalise
    word_in=word_in.upper()
    #If placed down
    if (not orientation):
        #For each letter in the inputted word
        for x in word_in:
            if (x == " "): #When a space is read skip a cell
                startPosTemp[1]=startPosTemp[1]+1
            else:
                #If cell is empty
                if (grid_in[startPosTemp[0]][startPosTemp[1]] == "+"):
                    #Assign current letter to element within grid passed
                    grid_in[startPosTemp[0]][startPosTemp[1]] = x
                    #Increment position by one
                    startPosTemp[1]=startPosTemp[1]+1
                else: #Cell is full
                    return False
    #If placed rightward
    else:
        for x in word_in:
            if (x == " "):
                startPosTemp[0]=startPosTemp[0]+1
            else:
                if (grid_in[startPosTemp[0]][startPosTemp[1]] == "+"):
                    grid_in[startPosTemp[0]][startPosTemp[1]] = x
                    startPosTemp[0]=startPosTemp[0]+1
                else:
                    return False
    #Successful placement
    return True
```

```
#Places word in sandbox grid, gathers surrounding words
#and checks words against the dictionary
def AreFormedWordsValid(grid_in,word_in,rot_in,coord_in,roundcounter_in):
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(coord_in)
    #Create copy of gamegrid
    sandboxGrid = CopyGrid(grid_in)
    #Write word to sandbox grid
    PlaceWord(rot_in,word_in,indexedCoord,sandboxGrid)
    #Grab all words formed during placement
    wordsToCheck = GrabAllFormedWords(sandboxGrid,word_in, rot_in,indexedCoord)
    #If not first placement
    if (roundcounter_in != 0):
        #If the placement is not connected to another
        if (len(wordsToCheck) == 1 and wordsToCheck[0].upper() == word_in.upper()):
            print("!Word MUST connect to another!")
            return False
    #Are all the words formed in the dictionary
    if (AreWordsInDictionary(wordsToCheck)):
        return True
    return False

#Checks if any tile overwrites another
def DoesPlacementOverwrite(grid_in,word_in,rot_in,coord_in):
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(coord_in)
    #Create copy of gamegrid
    tempGridCopy = CopyGrid(grid_in)
    #If word overwrites another
    if (not PlaceWord(rot_in,word_in,indexedCoord,tempGridCopy)):
        print("!You cannot overwrite tiles!")
        return True
    return False

#Determines if the player owns the tiles
#that are required to create the specified word
def DoesPlayerOwnTiles(word_in,inventory_in):
    #Create a true copy of inventory
    tempInv = inventory_in.copy()
    #Stores letters used as blanks
    blankLetters = []
    #For each letter in word
    for x in word_in.upper():
        #If not using a skip character
        if (x != " "):
            #If player has tile
            if (x in tempInv):
                #Remove first occurrence of tile
                del tempInv[tempInv.index(x)]
            #Player doesn't have tile
            else:
                #If player has a blank
                if (" " in tempInv):
                    #Remove first occurrence of blank
                    print(">Used blank for",x)
                    #Store tile used as blank
                    blankLetters.append(x)
                    del tempInv[tempInv.index(" ")]
                #Cannot make placement
                else:
                    print("!You do not have ",x, " to form this word!")
                    return [False]
    #Return successful and any blank tiles used
    return [True,blankLetters]
#Ensures the first placement passes through centre
#and is of length 2 or more
def DoesFirstWordMeetRequirements(grid_in,word_in,rot_in,coord_in):
    #Length check
    if (len(word_in) < 2):
        print("!The first word MUST be 2 or more characters in length!")
        return False
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(coord_in)
    #Create copy of gamegrid
    sandboxGrid = CopyGrid(grid_in)
    #Write word to sandbox grid
    PlaceWord(rot_in,word_in,indexedCoord,sandboxGrid)
    #Check if centre cell is populated
    if (sandboxGrid[7][7] == "+"):
        print("!The first word MUST pass through the centre!")
        return False
    return True
```

```
#Checks if a passed input is in the correct formatting e.g B9
#of correct type and is within the specified range for both axis
def ValidateCoord(coord_in):
    validCoordsX = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O"]
    #If input is two or three characters
    if (len(coord_in) <= 3 and len(coord_in) >= 2):
        #Get substrings
        firstChar = coord_in[0].upper() #Convert first element to uppercase
        secondChar = coord_in[1:] #Grab everything after the first element
        #If first character is within valid coordinates
        if (firstChar in validCoordsX):
            #Check if second value is an integer
            if (CheckIfInt(secondChar)):
                #If within range 1-15
                if (int(secondChar) >= 1 and int(secondChar) <= 15):
                    return True
    return False

#Takes in a validated coordinated e.g. G12
#and convert it to it's equivilent index based value
#e.g A1 = [0,0]
def ConvertCoordToIndex(coord_in):
    validCoordsX = ["A","B","C","D","E","F","G","H","I","J","K","L","M","N","O"]
    #Grab substrings again
    firstChar = coord_in[0].upper() #Convert to uppercase
    secondChar = coord_in[1:] #Grab everything after the first character
    #Get index position of firstChar within validCoordsX
    firstIndex = validCoordsX.index(firstChar)
    #Cast second substring as integer and take one
    secondIndex = int(secondChar)- 1
    #return indexes in list format
    return [firstIndex,secondIndex]

#Checks that the word is correctly formatted
#also detecting a turn forfeit
def ValidateWordInput():
    #Repeat until a valid input is given
    while True:
        #Retrieve word from user in uppercase
        wordAttempt = input("Enter the word to place\nor enter \"pass!\" to forfeit your turn\n").upper()
        #First check for forfeit command
        if (wordAttempt == "PASS!"):
            return Event.SKIPTURN
        #If string contains only spaces and alphabetic characters
        if (all(letter.isalpha() or letter.isspace() for letter in wordAttempt)):
            #If string doesnt only contain space/newline characters
            if (not all(letter.isspace() for letter in wordAttempt)):
                if (len(wordAttempt) <= 15): #If less than 15 chars
                    return wordAttempt #Return uppercase validword
                else:
                    print("!The word entered MUST be less than 15 characters!")
            else:
                print("!The word MUST contain atleast one letter!")
        else:
            print("!The word MUST only contain letters and spaces!")
#Gathers rotational value
#also detecting turn reattempt
def ValidateRotationInput():
    while True:
        orientation = input("Enter \"1\" for right, \"2\" for down\nor \"RE\" to reattempt your turn\n").upper()
        if (orientation == "RE"): #Reset inputs
            return Event.RESETTURN
        elif (orientation == "1"): #Rightward placement
            return True
        elif (orientation == "2"): #Downward placement
            return False
        else:
            print("!Invalid input, please try again!")

#Gathers and checks the coordinate
#also detecting turn reattempt
def ValidateCoordInput():
    while True:
        validCoord = input("Enter a coordinate in the format: \"A1\"\nor \"RE\" to reattempt your turn\n").upper()
        if (validCoord == "RE"): #Reset inputs
            return Event.RESETTURN
        #If coordinate is valid
        if (ValidateCoord(validCoord)):
            #Breaks loop and returns valid value
            return validCoord
        else:
            print("!Invalid coordinate!")
```

```
#Manager method, gathering each user input
#accounting for turn forfeits and reattempts
#returning successful inputs as a new instance of InputData class
def RetrieveInputs():
    #Gather word input or check for pass
    returnedWordInput = ValidateWordInput()
    #If entered pass, stop processing for this player
    if (returnedWordInput == Event.SKIPTURN):
        return Event.SKIPTURN
    #Otherwise continue
    #Gather rotation input and check for reset
    returnedRotInput = ValidateRotationInput()
    if (returnedRotInput == Event.RESETTURN):
        return Event.RESETTURN
    #Otherwise continue
    #Gather coordinate input and check for reset
    returnedCoordInput = ValidateCoordInput()
    if (returnedCoordInput == Event.RESETTURN):
        return Event.RESETTURN
    #Otherwise continue
    #Create and return new instance of input data
    return InputData(returnedWordInput,returnedRotInput,returnedCoordInput)

#The manager method used to call other methods,
#removing calls from the main game loop
#performing all the tasks required during a player's turn
def ProcessTurn(playerletters_in, gamegrid_in, roundcounter_in, bonusgrid_in,playerscore_in):
    while True:
        #Gather user input
        userInputs = RetrieveInputs()
        #Handling returns
        if (userInputs == Event.SKIPTURN):
            print(">Forfeiting turn")
            return [False]
        elif (userInputs == Event.RESETTURN):
            print(">Reattempting inputs")
            continue #Iterate loop again
        #If successful user return
        elif (type(userInputs) == InputData):
            #If first placement
            if (roundcounter_in == 0):
                #If word isnt correct length and doesnt pass through centre
                if (not DoesFirstWordMeetRequirements(gamegrid_in,userInputs.word,userInputs.rotation,userInputs.coord)):
                    continue #Iterate loop again
            #Otherwise, grab player word input data
            tileData = DoesPlayerOwnTiles(userInputs.word,playerletters_in)
            #If player owns all tiles required to place word
            if (tileData[0]):
                #If word doesn't overwrite another
                if (not DoesPlacementOverwrite(gamegrid_in,userInputs.word,userInputs.rotation,userInputs.coord)):
                    #If all formed words are valid and English
                    if (AreFormedWordsValid(gamegrid_in,userInputs.word,userInputs.rotation,userInputs.coord,roundcounter_in)):
                        #Placement is valid
                        break #Exit loop and process given inputs
    #Process the valid placement
    #Convert coordinate to index based coord
    indexedCoord = ConvertCoordToIndex(userInputs.coord)
    #Write word to main grid
    PlaceWord(userInputs.rotation,userInputs.word,indexedCoord,gamegrid_in)
    #Calculate the score earned during this placement
    earnedScore = CalculateScore(gamegrid_in,bonusgrid_in,userInputs.word,tileData,userInputs.rotation,indexedCoord)
    #Remove used tiles from inventory
    LetterDeductor(playerletters_in,userInputs.word)
    #Re-distribute tiles if available
    Dealer(playerletters_in)
    #Return as successful turn and give earned score
    return [True,earnedScore]

#Returns a copy of the grid passed
def CopyGrid(grid_in):
    return copy.deepcopy(grid_in)
```

```
#Using the given 2D array, draw it as a grid
#Displays the current state of the grid
#Assumes the grid is 15x15
#Also drawing sidebar
def DrawGrid(grid_in,bonus_in):
    print("") #Start new line
    #Store for sidebar text
    sidebarHelp = [" Letter Scoring:","[A]1  [B]3  [C]3  [D]2  [E]1  [F]4",
                   "[G]2  [H]4  [I]1  [J]8  [K]5  [L]1","[M]3  [N]1  [O]1  [P]3  [Q]10 [R]1",
                   "[S]1  [T]1  [U]1  [V]4  [W]4  [X]8","[Y]4  [Z]10 [ ]0","Bonus types:",
                   "TWS:6  DWS:2  TLS:3  DLS:1","Helpful Tips:",
                   "1. The coordinate is the first letter's", "position.",
                   "2. When using a placed letter, use a [space]","in its place.",
                   "3. When using a blank, enter the letter","as if you have it."]
    #Output top row coords
    print("   A B C D E F G H I J K L M N O", end = "")
    #For each item in board
    for y in range(15):
        if (y < len(sidebarHelp)): #If item in sidebarHelp
            #Print with space to the left
            print(" "*2+sidebarHelp[y],end="")
        for x in range(15):
            if (x % 15 == 0): #If at end of the row
                print("") #Start new line
                print("%2d" % (y+1,), end = " ")
            #If bonus contents is blank
            if (bonus_in[y][x] == " "):
                #Print game grid
                print(grid_in[x][y], end = " ")
            else: #If bonus contents is populated
                #Print special character
                print(bonus_in[y][x], end = " ")
    print("") #Start new line

#Distribute 7 random tiles from the bag to each player,
#returning a 2D array containing each players starting inventory
def SetupLetters(numPlayers_in):
    playerInventoriesTemp=[] #Temp storage for all player inventories (2D)
    for x in range(numPlayers_in): #For each player
        currentPlayerInventory=[] #Temp storage for THIS player's inventory
        for y in range(7): #For each letter to give
            #Pick random index of letter within total bag size
            #Inclusive,exclusive
            randomIndex = random.randint(0,len(availableTiles)-1)
            #Assign current player that random tile
            currentPlayerInventory.append(availableTiles[randomIndex])
            #Remove tile from bag
            del availableTiles[randomIndex]
        #Store created inventory
        playerInventoriesTemp.append(currentPlayerInventory)
    #Return 2D array of inventories
    return playerInventoriesTemp

#Gather and return a list of player names to be referenced each turn
def SetupPlayers(numPlayers_in):
    #Temporary local variable
    names=[]
    #For each player
    for x in range(numPlayers_in):
        #Gather user name (unvalidated)
        newPlayerName = input("Enter player "+str(x+1)+ " name ")
        #Add input to local variable
        names.append(newPlayerName)
    return names

#Checks if a value is an integer
def CheckIfInt(value_in):
    try:
        int(value_in)
    except ValueError:
        return False
    return True

#Retrieve and return a valid number of players from the user
def GetNumPlayers():
    #Repeatedly ask for an input, until it exited
    while True:
        #Gather input
        numPlayers = input("Enter the number of players, 1-4 ")
        #If value is an integer
        if (CheckIfInt(numPlayers)):
            numPlayers = int(numPlayers) #Convert to int
            #If value is in range
            if (numPlayers > 0 and numPlayers < 5):
                return numPlayers #Exit loop and return value
            else:
                print("!Number MUST be between 1-4!")
        else:
            print("!You MUST enter a number!")
```

```
#Read dicitonary file and convert to a Python dictionary
def CreateDictionary():
    #Read contents of file
    file = open("words_dictionary.json","r")
    #Literally convert string to data type
    dictionary = ast.literal_eval(file.read())
    print("The dictionary has",len(dictionary), "words")
    return dictionary

##Classes##
#Stores "constant" reference values for event types
class Event:
    SKIPTURN = 1
    RESETTURN = 2
#Stores data about the user inputs
class InputData:
    #Constructor, assign values of each attribute
    def __init__(self,word,rotation,coord):
        self.word = word
        self.rotation = rotation
        self.coord = coord

##Main program##
##Init##
#Bag contents
availableTiles = [" "," ","E","E","E","E","E","E","E","E","E","E","E","E",
                  "I","I","I","I","I","I","I","I","I","I","I","I",
                  "O","O","O","O","O","O","O","O","N","N","N","N","N","N","R","R","R","R","R","R",
                  "T","T","T","T","T","T","L","L","L","L","S","S","S","S",
                  "U","U","U","U","D","D","D","D","G","G","G","B","B","C","C",
                  "M","M","P","P","F","F","H","H","V","V","W","W","Y","Y","K","J","X","Q","Z"]

#Tile score reference dictionary
tileScores = {" ":0,"A":1,"B":3,"C":3,"D":2,"E":1,"F":4,"G":2,"H":4,
              "I":1,"J":8,"K":5,"L":1,"M":3,"N":1,"O":1,"P":3,"Q":10,
              "R":1,"S":1,"T":1,"U":1,"V":4,"W":4,"X":8,"Y":4,"Z":10}
#Bonus grid layout + centre marker
bonusGrid=[["6"," "," ","1"," "," "," ","6"," "," "," ","1"," "," ","6"],
           [" ","2"," "," "," ","3"," "," "," ","3"," "," "," ","2"," "],
           [" "," ","2"," "," "," ","1"," ","1"," "," "," ","2"," "," "],
           ["1"," "," ","2"," "," "," ","1"," "," "," ","2"," "," ","1"],
           [" "," "," "," ","2"," "," "," "," "," ","2"," "," "," "," "],
           [" ","3"," "," "," ","3"," "," "," ","3"," "," "," ","3"," "],
           [" "," ","1"," "," "," ","1"," ","1"," "," "," ","1"," "," "],
           ["6"," "," ","1"," "," "," ","0"," "," "," ","1"," "," ","6"],
           [" "," ","1"," "," "," ","1"," ","1"," "," "," ","1"," "," "],
           [" ","3"," "," "," ","3"," "," "," ","3"," "," "," ","3"," "],
           [" "," "," "," ","2"," "," "," "," "," ","2"," "," "," "," "],
           ["1"," "," ","2"," "," "," ","1"," "," "," ","2"," "," ","1"],
           [" "," ","2"," "," "," ","1"," ","1"," "," "," ","2"," "," "],
           [" ","2"," "," "," ","3"," "," "," ","3"," "," "," ","2"," "],
           ["6"," "," ","1"," "," "," ","6"," "," "," ","1"," "," ","6"]]

#Read and store dictionary
dictionary = CreateDictionary()
#Gather number of players
numPlayers = GetNumPlayers()
#Give each player a name
playerNames = SetupPlayers(numPlayers)
#Give each player a set of 7 tiles
playerInventories = SetupLetters(numPlayers)
#Create an entry for each player
playerScores = [0]*numPlayers
#Construct 2D list of size 15x15, assigning each element as a +
gameGrid=[["+"]*15 for i in range(15)]
#Display initial state of the grid
DrawGrid(gameGrid,bonusGrid)

#Declare required game loop variables
currentTurn = 0
roundCounter = 0
passCounter = 0
successfulEndType = False
```

```
##Game Loop##
while True:
    #If at last player, return back to first player index
    if (currentTurn % numPlayers == 0):
        currentTurn = 0
    #Display user data based on current turn
    print("It's " + playerNames[currentTurn] + "'s turn!")
    #Display current users inventory, formatted
    print("-------Your Letters-------")
    print("| "+" | ".join(playerInventories[currentTurn])+" |")
    #Call turn manager method
    turnOutcome = ProcessTurn(playerInventories[currentTurn],gameGrid,roundCounter,bonusGrid,playerScores[currentTurn])
    #If placement occurs, increment round and modify score
    if (turnOutcome[0]):
        #Tracks number of successful placements
        roundCounter += 1
        #Increment player's score value
        playerScores[currentTurn] += turnOutcome[1]
        #Reset forfeit counter
        passCounter = 0
    else: #User forfeited turn
        passCounter += 1 #Increment forfeit counter
    #Draw new state of the grid
    DrawGrid(gameGrid,bonusGrid)
    #If player has used all their tiles, end the game
    if (len(playerInventories[currentTurn]) == 0):
        print(playerNames[currentTurn]+" has used all of their tiles!\n")
        successfulEndType=True
        break
    #If everybody has forfeit their turn
    if (passCounter == numPlayers):
        print(">Nobody can make a placement!\n")
        successfulEndType=False
        break
    #Waits for input to slow down pace of game
    input("Tap enter\n")
    #Next player index
    currentTurn+=1
##Game over##
print("==Game over==")
#Adjust final scores depending on outcome
adjustedScores = CalculateEndScores(successfulEndType,playerInventories,playerScores,playerNames,currentTurn)
#Determine the winner of the game
DetermineWinner(adjustedScores,playerNames)
```

Gameplay output

To review the final product, and to test the robustness, and execution of each sub system that has been developed. The most suitable method of testing, would be to play through an entire game. As, this will insure that all the main systems are being used, and that the product is capable of performing its task, in being an accurate recreation of the original board game, Scrabble. Allowing the comparison between the final product, and the initial criteria to be made.

To reiterate what was stated before. Due to the random nature of this game, there cannot be predefined test data, meaning, that game states are not repeatable. It is because of this limitation, that the most realistic, and representative use case for this program, is to simply play it, from start to finish.

```
The dictionary has 370101 words
Enter the number of players, 1-4 3
Enter player 1 name Luc
Enter player 2 name Will
Enter player 3 name Tom

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + + 0 + + + 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 + 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + 3 + + + 3 +   position.
11 + + + + 2 + + + + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
It's Luc's turn!
-------Your Letters-------
| I | U | E | Q | V | T | E |
Enter the word to place
or enter "pass!" to forfeit your turn
quite
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
g8

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 + 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + 3 + + + 3 +   position.
11 + + + + 2 + + + + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter
```

```
It's Will's turn!
-------Your Letters-------
| Z | O | S | R | E | D | T |
Enter the word to place
or enter "pass!" to forfeit your turn
ested
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
j8
!You do not have  E  to form this word!
Enter the word to place
or enter "pass!" to forfeit your turn
st
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
i8
! QUITEST  is not a valid word!
Enter the word to place
or enter "pass!" to forfeit your turn
pass!
>Forfeiting turn

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 + 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + 3 + + + 3 +   position.
11 + + + + 2 + + + + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter
```

```
It's Tom's turn!
-------Your Letters-------
| N | I | I | M | T | O | D |
Enter the word to place
or enter "pass!" to forfeit your turn
imid
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
j9
>Triple letter bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 + 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + 3 + + + 3 +   position.
11 + + + + 2 + + + + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Luc's turn!
-------Your Letters-------
| V | E | L | I | E |   | V |
Enter the word to place
or enter "pass!" to forfeit your turn
elve
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
j12
!You cannot overwrite tiles!
Enter the word to place
or enter "pass!" to forfeit your turn
elve
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
k12
>Double word bonus!
```

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 + 1 I + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + M + + + 3 +   position.
11 + + + + 2 + + + + I 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + D E L V E 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Will's turn!
-------Your Letters-------
| Z | O | S | R | E | D | T |
Enter the word to place
or enter "pass!" to forfeit your turn
r
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
o12
>Double letter bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 + 1 I + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + M + + + 3 +   position.
11 + + + + 2 + + + + I 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Tom's turn!
-------Your Letters-------
| N | T | O | C | Y | I | F |
Enter the word to place
or enter "pass!" to forfeit your turn
int
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
```

```
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
re
>Reattempting inputs
Enter the word to place
or enter "pass!" to forfeit your turn
c t
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
h7

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 C 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + M + + + 3 +   position.
11 + + + + 2 + + + + I 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Luc's turn!
-------Your Letters-------
| I |   | V | R | G | I | I |
Enter the word to place
or enter "pass!" to forfeit your turn
vi id
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
m10
>Used blank for D
>Double word bonus!
```

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 C 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + M + + V 3 +   position.
11 + + + + 2 + + + + I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + D 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Will's turn!
-------Your Letters-------
| Z | O | S | E | D | T | E |
Enter the word to place
or enter "pass!" to forfeit your turn
ted
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
h10
>Double letter bonus!
```

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 C 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V 3 +   position.
11 + + + + 2 + + E + I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + D 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Tom's turn!
-------Your Letters-------
| N | O | Y | I | F | I | U |
Enter the word to place
or enter "pass!" to forfeit your turn
fun
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
j14
>Triple letter bonus!
```

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 C 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V 3 +   position.
11 + + + + 2 + + E + I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Luc's turn!
-------Your Letters-------
| R | G | I | B | N | N | E |
Enter the word to place
or enter "pass!" to forfeit your turn
er
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
n14
>Double word bonus!
```

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 C 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V 3 +   position.
11 + + + + 2 + + E + I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Will's turn!
-------Your Letters-------
| Z | O | S | E | O | H | S |
Enter the word to place
or enter "pass!" to forfeit your turn
o
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
o15
!Invalid coordinate!
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
o
!Invalid coordinate!
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
o15
>Triple word bonus!
```

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 C 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V 3 +   position.
11 + + + + 2 + + E + I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Tom's turn!
-------Your Letters-------
| O | Y | I | I | N | W | I |
Enter the word to place
or enter "pass!" to forfeit your turn
in
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
j6
! INTIMID  is not a valid word!
Enter the word to place
or enter "pass!" to forfeit your turn

!The word MUST contain atleast one letter!
Enter the word to place
or enter "pass!" to forfeit your turn
nit
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
```

```
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
e11
! NITE  is not a valid word!
Enter the word to place
or enter "pass!" to forfeit your turn
y t
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
g11

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 C 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V 3 +   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Luc's turn!
-------Your Letters-------
| G | I | B | N | N | D | I |
Enter the word to place
or enter "pass!" to forfeit your turn
binn d
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
k4
>Double word bonus!
```

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N + + 3 +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V 3 +   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Will's turn!
-------Your Letters-------
| Z | S | E | O | N | S | E |
Enter the word to place
or enter "pass!" to forfeit your turn
ose
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
l6
>Triple letter bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V 3 +   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter
```

```
It's Tom's turn!
-------Your Letters-------
| O | I | I | N | W | M | U |
Enter the word to place
or enter "pass!" to forfeit your turn
in
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
n10
>Triple letter bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Luc's turn!
-------Your Letters-------
| G | I | R | D | E | C | G |
Enter the word to place
or enter "pass!" to forfeit your turn
ride
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
14
>Double word bonus!
>Double letter bonus!
```

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Will's turn!
-------Your Letters-------
| Z | E | S | H | O | B | Y |
Enter the word to place
or enter "pass!" to forfeit your turn
h a
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
m3
>Double word bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + H + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Tom's turn!
-------Your Letters-------
| O | I | W | M | U | O | T |
Enter the word to place
or enter "pass!" to forfeit your turn
mot
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
o1
>Triple word bonus!
```

Luc Heaton

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Luc's turn!
-------Your Letters-------
| G | C | G | R | N | N | L |
Enter the word to place
or enter "pass!" to forfeit your turn
r
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
i8
! QUITER  is not a valid word!
Enter the word to place
or enter "pass!" to forfeit your turn
pass!
>Forfeiting turn

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Will's turn!
-------Your Letters-------
| Z | H | O | B | Y | E | U |
Enter the word to place
or enter "pass!" to forfeit your turn
zo
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
j2
>Triple letter bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + O + 2 O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Luc's turn!
-------Your Letters-------
| G | C | G | R | N | N | L |
Enter the word to place
or enter "pass!" to forfeit your turn
pass!
>Forfeiting turn

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + O + 2 O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Will's turn!
-------Your Letters-------
| Z | H | O | B | Y | I | J |
Enter the word to place
or enter "pass!" to forfeit your turn
ji
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
o8
>Triple word bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Tom's turn!
-------Your Letters-------
| I | W | U | O | P | P |   |
Enter the word to place
or enter "pass!" to forfeit your turn
poo
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
l1
>Used blank for O
>Double letter bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + 2 O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Tom's turn!
-------Your Letters-------
| I | W | U | P | L | E | L |
Enter the word to place
or enter "pass!" to forfeit your turn
upwell
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
e1
! LZ  is not a valid word!
Enter the word to place
or enter "pass!" to forfeit your turn
pass!
>Forfeiting turn

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + 2 O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter
```

```
It's Luc's turn!
-------Your Letters-------
| G | C | G | R | N | N | L |
Enter the word to place
or enter "pass!" to forfeit your turn
c
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
n2
>Double word bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 + 3 + + + 3 + T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Will's turn!
-------Your Letters-------
| H | B | Y | E | U | O | O |
Enter the word to place
or enter "pass!" to forfeit your turn
boot
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
re
>Reattempting inputs
Enter the word to place
or enter "pass!" to forfeit your turn
boo
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
e10
>Triple letter bonus!
```

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 + 3 + + B O O T + M + + V I N   position.
11 + + + + 2 + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + D + D E L V E R   in its place.
13 + + 2 + + + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Tom's turn!
-------Your Letters-------
| I | W | U | P | L | E | L |
Enter the word to place
or enter "pass!" to forfeit your turn
lew
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
e11
>Double word bonus!
```

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 + 3 + + B O O T + M + + V I N   position.
11 + + + + L + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 1 + + 2 E + + D + D E L V E R   in its place.
13 + + 2 + W + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter
```

```
It's Luc's turn!
-------Your Letters-------
| G | G | R | N | N | L | E |
Enter the word to place
or enter "pass!" to forfeit your turn
genr
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
a12
>Double letter bonus!
>Double word bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 + 3 + + B O O T + M + + V I N   position.
11 + + + + L + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 G E N R E + + D + D E L V E R   in its place.
13 + + 2 + W + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Will's turn!
-------Your Letters-------
| H | Y | E | U | E | I | R |
Enter the word to place
or enter "pass!" to forfeit your turn
hu e
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
a10

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + 1 + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 H 3 + + B O O T + M + + V I N   position.
11 U + + + L + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 G E N R E + + D + D E L V E R   in its place.
13 E + 2 + W + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter
```

```
It's Tom's turn!
-------Your Letters-------
| I | U | P | L | G | K | O |
Enter the word to place
or enter "pass!" to forfeit your turn
poki g
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
c8
>Double letter bonus!
>Double word bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + P 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + O + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 H 3 K + B O O T + M + + V I N   position.
11 U + I + L + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 G E N R E + + D + D E L V E R   in its place.
13 E + G + W + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Luc's turn!
-------Your Letters-------
| G | N | L | R | T | X | S |
Enter the word to place
or enter "pass!" to forfeit your turn
x
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
n13
! EXE  is not a valid word!
Enter the word to place
or enter "pass!" to forfeit your turn
pass!
>Forfeiting turn
```

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + P 1 + + Q U I T E 1 + + J   Helpful Tips:
 9 + + O + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 H 3 K + B O O T + M + + V I N   position.
11 U + I + L + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 G E N R E + + D + D E L V E R   in its place.
13 E + G + W + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Will's turn!
-------Your Letters-------
| Y | E | I | R | I | F | S |
Enter the word to place
or enter "pass!" to forfeit your turn
re s
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
a8
>Triple word bonus!
>Double letter bonus!
>No tiles left in bag to redistribute

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 + + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 R E P S + + Q U I T E 1 + + J   Helpful Tips:
 9 + + O + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 H 3 K + B O O T + M + + V I N   position.
11 U + I + L + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 G E N R E + + D + D E L V E R   in its place.
13 E + G + W + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter
```

```
It's Will's turn!
-------Your Letters-------
| Y | I | I | F |
Enter the word to place
or enter "pass!" to forfeit your turn
pass!
>Forfeiting turn

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + L 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + U + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 E + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 R E P S + + Q U I T E 1 + + J   Helpful Tips:
 9 + + O + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 H 3 K + B O O T + M + + V I N   position.
11 U + I + L + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 G E N R E + + D + D E L V E R   in its place.
13 E + G + W + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Tom's turn!
-------Your Letters-------
| W |
Enter the word to place
or enter "pass!" to forfeit your turn
w
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
n15
>No tiles left in bag to redistribute

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + L 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + U + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 E + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 R E P S + + Q U I T E 1 + + J   Helpful Tips:
 9 + + O + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 H 3 K + B O O T + M + + V I N   position.
11 U + I + L + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 G E N R E + + D + D E L V E R   in its place.
13 E + G + W + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + W O
Tom has used all of their tiles!

==Game over==
Tom  is the winner!
with  141  points!
>>>
```

```
It's Tom's turn!
-------Your Letters-------
| U | L | E | W |
Enter the word to place
or enter "pass!" to forfeit your turn
lue
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
d5
>No tiles left in bag to redistribute

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + L 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + U + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 E + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 R E P S + + Q U I T E 1 + + J   Helpful Tips:
 9 + + O + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 H 3 K + B O O T + M + + V I N   position.
11 U + I + L + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 G E N R E + + D + D E L V E R   in its place.
13 E + G + W + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter

It's Luc's turn!
-------Your Letters-------
| G | N | L | R | T | X | S |
Enter the word to place
or enter "pass!" to forfeit your turn
pass!
>Forfeiting turn

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + L 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + U + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 E + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 R E P S + + Q U I T E 1 + + J   Helpful Tips:
 9 + + O + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 H 3 K + B O O T + M + + V I N   position.
11 U + I + L + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 G E N R E + + D + D E L V E R   in its place.
13 E + G + W + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + O
Tap enter
```

Evaluation

Although the final testing of the product was not a systematic check of each sub system, it did test the function of each system as it was used during a normal use case. Allowing the review of the usability and robustness of the program, by its capability to portray an accurate recreation of the Scrabble board game, in a text-based format.

Sequentially running through the finalised success criteria, comparing each, to the implementation, or lack of, within the final product. As an extra consideration, due to the repetitive, turn-based nature of the game, I will copy the first/notable instance(s) of each output, that illustrates the extent to which each criterion is met. In addition to this, some of the criteria cannot be shown within this final gameplay output (above), as each possible scenario was not explored (in this test), due to the fact that there are so many outcomes. Although, evidence for the criteria may be shown by displaying outputs of the code that explore these scenarios, unless stated otherwise.

The game is played by up to four players.

```
Enter the number of players, 1-4 no
!You MUST enter a number!
Enter the number of players, 1-4 77
!Number MUST be between 1-4!
Enter the number of players, 1-4 1
Enter player 1 name Luc
```

When the game begins, the first input that is required from the user, is the number of players that will take-part in the game. It has been developed so that the value entered, must be within the given range. Any value that is outside of this range, or is not an integer will be rejected, and a meaningful message is given. In the example above, the input of "no", is responded with the message "!You MUST enter a number!", whereas the out-of-range value "77", is responded with "!Number MUST be between 1-4!". Giving specific indications to the user, as to the problem with their input.

This point has been fully met, as it performs its main task, in limiting the number of players, as well as successfully accounting for any user input.

The gameplay should not change depending on the number of players.

```
Enter the number of players, 1-4 4
Enter player 1 name Luc
Enter player 2 name Will
Enter player 3 name Tom
Enter player 4 name Bob
...
It's Luc's turn!
-------Your Letters-------
| U | D |   | M | C | O | N |
Enter the word to place
or enter "pass!" to forfeit your turn
MODE
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
H8
>Used blank for E

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + + M + + + 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 O 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + D + 3 + + + 3 +   position.
11 + + + + 2 + + E + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Will's turn!
-------Your Letters-------
| B | E | E | G | L | O | Y |
Enter the word to place
or enter "pass!" to forfeit your turn
```

```
Enter the number of players, 1-4 1
Enter player 1 name Luc
...
It's Luc's turn!
-------Your Letters-------
| L | O |   | U | E | D | E |
Enter the word to place
or enter "pass!" to forfeit your turn
duel
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
h8

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + + D + + + 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 U 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + E + 3 + + + 3 +   position.
11 + + + + 2 + + L + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Luc's turn!
-------Your Letters-------
| O |   | E | I | Q | N | M |
Enter the word to place
or enter "pass!" to forfeit your turn
```

This point is shown by the turn-based gameplay of the game, which is independent of the number of players entered initially. As, for any number of players (within the given range) the game will either continue to the next player, if there is more than one player. Or will return back to the first player, upon reaching the last. So, in the case of a single player, after taking their turn, it will then become their turn again. But, if this single player is to forfeit their turn, the game will end and they will be announced as the winner, alongside a score value.

I am confident that this point has been fully met, with the gameplay remaining the same for any number of players. As the systems do not alter the gameplay depending on it, they only account for it.

The game will be turn-based.

```
Enter the number of players, 1-4 2
Enter player 1 name Luc
Enter player 2 name Will
...
It's Luc's turn!
-------Your Letters-------
| O | I | Z | I | N | I | O |
...
It's Will's turn!
-------Your Letters-------
| P | T | M | O | Y | P | E |
...
It's Luc's turn!
-------Your Letters-------
| I | Z | I | I | O | T | I |
```

Similarly expressed above, this point has been fully met. Illustrating that after each successive player's turn, it is then the next. Returning back to the first player after reaching the last. This can also be seen by tracking of each players alias (or name), as well as each player's inventory, that is specific to them.

The only issue with this solution, is due to the limitation caused by using the same local machine for each player. As, in the original game of Scrabble, individual players would sit facing each other, so that other player's tiles cannot be seen, this adds a level of tactics to the game, as a player cannot predict your placement. However, as this is game does not revolve around hidden information, this does not cause too much of an issue. Although, a solution to this issue could be a command that is entered on a player's turn, that displays their tiles, allowing the other player(s) to turn around. Despite this minor issue, the game is completely turn-based, and so this point has been fully fulfilled.

Each player should be allowed to assign themselves an alias.

```
Enter the number of players, 1-4 3
Enter player 1 name Luc
Enter player 2 name Tom
Enter player 3 name Will
...
It's Luc's turn!
-------Your Letters-------
| E | S | G | U | O | U | U |
Enter the word to place
or enter "pass!" to forfeit your turn
```

This point was implemented fully, allowing for the number of players entered prior, to enter any identifier for themselves. Which is then referenced throughout the game, when it is their turn, or if they win the game. Although, if I were to redesign this program, I would implement the storage of player data differently. In this current state, each of the user's values that are tracked throughout the game, such as: player names, player inventories and player scores, using separate lists, and referenced by shared indexes.
If an issue were to occur with any of this data, or if more than four pieces of data are required, it would be fairly difficult to debug the issue. The seemingly obvious solution to this problem, would be to incorporate a class that stores an attribute for all three data points, removing the requirement for list indexing, making the program more maintainable, if it were to scale.

There should be a character limit for the aliases.

```
Enter the number of players, 1-4 1
Enter player 1 name Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua. Ut enim ad minim veniam, quis nostru
...
It's Lorem ipsum dolor sit amet, consectetur
cididunt ut labore et dolore magna aliqua. U
itation ullamco laboris nisi ut aliquip ex e ...
 in reprehenderit in voluptate velit esse ci
pteur sint occaecat cupidatat non proident,
anim id est laborum.'s turn!
-------Your Letters-------
| N | D | E | I | S | R | D |
Enter the word to place
or enter "pass!" to forfeit your turn
```

This was a mindless mistake that should have been avoided. However, during development, this limitation to length of player names was overlooked, and so, it allows a player to enter a name of any length, which is then displayed to each time it is their turn. Showing that this point has not been met. In further developments this could simply be resolved by including a condition during the user input, similar to that of other methods, checking the length of the input. Requesting that input is below a certain range.

The interface for user inputs must be validated.

```
Enter the word to place
or enter "pass!" to forfeit your turn
%$^hi^"
!The word MUST only contain letters and spaces!
Enter the word to place
or enter "pass!" to forfeit your turn

!The word MUST contain atleast one letter!
Enter the word to place
or enter "pass!" to forfeit your turn
h3lp
!The word MUST only contain letters and spaces!
Enter the word to place
or enter "pass!" to forfeit your turn
line
```

```
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
0
!Invalid input, please try again!
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
```

```
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
z1
!Invalid coordinate!
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
a16
!Invalid coordinate!
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
%1
!Invalid coordinate!
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
a%
!Invalid coordinate!
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
g7
```

For a program that revolves around user input, this was a necessity. Each of the three user inputs, must be validated before being passed to other systems. For example, the implementation of the word input must confirm that:

- The input is only alphabetic characters and spaces.
- The input is constructible using the tiles that are available to the player, including blanks.
- The word does not overwrite another.
- The word formed and any resultant words are all valid English words.
- The word connects to another (unless it is the first word).
- The word does not exit the board.
- The first word must pass through the centre cell.

It can be noted that this validation is fairly comprehensive, and many checks are required before a word can be deemed valid to place. Even so, I have implemented all of these checks for all the possible outcomes of each user input (including coordinate and rotation inputs), giving meaningful messages in the case that one of the checks is not met. Making the usability of the program significantly clearer and easier to use.

Due to the great extent to which this method of validation has been implemented, this criterion has been fully met, with the extra added benefit of usability through responsive error messages, also, allowing invalid inputs to be reattempted until a valid input is given, or otherwise.

The capitalisation of user inputs should have no effect.

```
Enter the word to place
or enter "pass!" to forfeit your turn
pEn
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
h8

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + + P + + + 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 E 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + N + 3 + + + 3 +   position.
11 + + + + 2 + + + + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter
```

Pairing with the statement above, it became a standard during the development of this project to normalise user inputs to uppercase, with the exception of dictionary checking, as the entries were in lowercase. This meant that independent of the capitalisation entered by the user, the expected output would be shown. Also, making the usability of the program simpler, as there is no requirement to the user to consider this extra element of input. Illustrating that this point was fully met.

A player should be able to reattempt their inputs, before their placement has been made.

```
It's Luc's turn!
-------Your Letters-------
| E | T | V | C | P | N | D |
Enter the word to place
or enter "pass!" to forfeit your turn
pet
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
re
>Reattempting inputs
Enter the word to place
or enter "pass!" to forfeit your turn
vet
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
re
>Reattempting inputs
Enter the word to place
or enter "pass!" to forfeit your turn
pet
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
h8
```

This felt like a necessary feature during development, as it is very common for an accidental input. Whereby a user would like to change their original input. This was implemented as a command that can be entered at the last two stages of user input, requesting the user inputs again, which can also be achieved any number of times, shown within the output above. This implementation was also an improvement in usability, making the inputs less concrete, and allowing a player to be more flexible with their decision making, without worry of mistaken inputs. Because of its intuitive implementation, alongside the other user inputs, I would say that this criterion was very successful, and has been fully met.

LUC HEATON

A player should be able to forfeit their turn, allowing the next player to play.
Where each player has forfeited their turn consecutively, the game should end.

```
It's Will's turn!
-------Your Letters-------
| L | M |   | T | Y | E | X |
Enter the word to place
or enter "pass!" to forfeit your turn
pass!
>Forfeiting turn

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + + P E T + 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 + 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + 3 + + + 3 +   position.
11 + + + + 2 + + + + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter

It's Luc's turn!
-------Your Letters-------
| V | C | N | D | R | O | R |
Enter the word to place
or enter "pass!" to forfeit your turn
pass!
>Forfeiting turn
...
>Nobody can make a placement!

==Game over==
Luc  is the winner!
with  -8  points!
>>>
```

Due to the random nature of distributions in the game, it is very possible that a user is simply unable to make a placement, because of this, it was crucial that the implementation of a turn forfeit was included. It works in a similarly intuitive way, to the reattempting of inputs. By acting as a command which is enterable during the first stage of user input (word input), the reasoning behind limiting its usage to only the word input was done, to reduce the amount of text that is displayed each turn, and also because it is possible to return to the first stage of input, using the reattempt inputs method above.

This method was built upon, with the addition of forfeiting to end the game. Allowing the game to be ended, after each successive player has entered this command, before scoring and announcing the winner as usual. This also felt like a necessary addition to the usability of the program, as it allows the game to end when players have had enough (as Scrabble can last a long time), or in the case that no player can make a placement. In the example output above, it can be noted that after the two players have successively forfeited their turns the game is ended and the winner is announced.

Within this example, the slight issue can be seen. As this method allows the game to be ended at any point during the game, meaning that it is therefore very possible to end the game, before players have gained a great enough score, where their remaining tile scores, minus their total score is equal to a negative number. As much as this is an unwanted outcome, it is justified as it is correct. However, this could be modified in post-development, to either score the players differently in this scenario, or to simply limit their score values to be greater than zero. Other than this very minor issue, both points have been fully implemented.

There will be a finite number of tiles available to use throughout the game.

```
It's Luc's turn!
-------Your Letters-------
| O | L | H | K | Z | O | G |
Enter the word to place
or enter "pass!" to forfeit your turn
zoo
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
h8
There are 80 tiles left
```

```
It's Will's turn!
-------Your Letters-------
| N | I | O | R | I | H | E |
Enter the word to place
or enter "pass!" to forfeit your turn
rhin
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
d10
>Triple letter bonus!
There are 77 tiles left
```

```
It's Tom's turn!
-------Your Letters-------
| U | L | E | W |
Enter the word to place
or enter "pass!" to forfeit your turn
lue
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
d5
>No tiles left in bag to redistribute
```

To accurately portray Scrabble, it is required that there is a finite/limited number of tiles that can be used by players throughout the game. This should be obvious due to the also limited number of cells on the board, meaning that it would be impossible to end the game without forfeiting. Due to the fact that the number of tiles, within the bag, is not tracked in the final solution, I have included a debug message to show its decrease after each placement, shown in the outputs above.

After each successful placement, the number of tiles used during the placement is then replaced, removing them from the finite sized "bag", for example, the three-letter placement of the word "zoo", deducts the three tiles used from the player, replacing them with random tiles from the bag, shown by the decrement of three. Once the bag becomes empty, replacement tiles are no longer distributed, and an additional message is shown, explaining to the player(s) why they are no longer receiving more tiles. Allowing the game to come to a end. This feature was fully implemented, with no additional issues.

Each tile has an associate tile score.

```
It's Luc's turn!
-------Your Letters-------
| I | T | P | S | T | O | S |
Enter the word to place
or enter "pass!" to forfeit your turn
stops
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
h8
Associate tile scores:  [1, 1, 1, 3, 1]
>Double letter bonus!

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + + S + + + 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 T 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + O + 3 + + + 3 +   position.
11 + + + + 2 + + P + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + S + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter
```

The solution to this issue, was the implementation of a dictionary. Where each tile (alphabetic characters and a blank), was a key with an associate score value as its pair. This allowed a lookup function to be created, which successfully iterates through the user's input, gathering the base tile scores for each tile. Again, similarly to above this cannot be seen in the main program, so to illustrate its correct usage, I have added a debug message stating the equivalent tile scores of the user's inputs (in-order). From the output above, the word "stops" is shown to have individual tile scores of "1,1,1,3,1", which, when checked against the helpful side bar I produced, is correct. In the case that a blank tile is used, its associate score would equal zero. This method performs its task successfully, producing an output which is then used in the scoring of a placement.

Blank tiles that have no score value, but can imitate any alphabetic character.

```
It's Luc's turn!
-------Your Letters-------
| K | L | I | I | O | E |   |
Enter the word to place
or enter "pass!" to forfeit your turn
liked
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
g8
>Used blank for D
Associate tile scores:  [1, 1, 5, 1, 0]

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + L I K E D 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 + 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + 3 + + + 3 +   position.
11 + + + + 2 + + + + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
Tap enter
```

The methods used to create this outcome, work correctly. From the example above, the program has detected the usage of a blank for the letter D, which is then placed on the board. Although the solution developed initially performed its task well, in detecting the usage of a blank(s) and successfully removing them from a player's inventory.

The requirement for scoring brought some issues, firstly, the poorly implemented adaptation to one of the methods, that was required to locate and score the blank tile correctly, within the user's placement. The second, more notable issue, which is caused by a limitation in the method of using blanks. Is that, upon using a blank tile it is correctly scored as zero. However, if and when a user decides to use that blank tile (imitating another tile) as an existing tile on the board, the program will treat it as-if it is a regular tile, scoring it incorrectly. This could potentially alter the result of a game, in the case that a blank is used for a high scoring letter, however due to the fairly low chance of a blank being drawn (1 in 50), and the odds of the blank being used by another placement, as well as the chance of the tile score being significant enough to even be noticed. Due to its fairly minimal impact on the game, I did not allocate the time to resolve the issue. It is because of this issue, that this criterion is partially met, and not fully.

If I were to revisit this project, my solution to the former issue would be to incorporate a method of tracking the location of blanks within the game grid. That way, it would be possible to determine where a tile should be treated as a blank, or as a regular tile.

There should be bonus tiles distributed around the game board.
The display will show the current state of the game, including placed words and bonus tiles.
Instructions to help the player understand the method of play.

```
   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + 1 + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + L I K E D 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 + 1 + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + + 3 + + + 3 +   position.
11 + + + + 2 + + + + + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
```

For a text-based representation of the game board, displaying all the necessary information required by the player. I feel as-though this is a very good display of the current state of the game. Displaying the coordinates required to make a placement, the distribution of bonus tiles, as well as existing placements. The area where this method of displaying bonuses fails, is the usage of characters to indicate bonus types, making the board slightly harder to interpret, affecting usability. However, this issue of understanding has been resolved to the best of my ability by including a helpful key, which is displayed alongside the board.
As much as this solution to the less-than-ideal display of bonus tiles fulfils the explanation of the characters, it does not quite meet the clear distribution of bonuses that I would have liked, so this point will be marked as partially complete.

For the second point, regarding the display of the board. I am confident that due to the limitations of a text-based display, that can only use one colour, my solution achieves its purpose very well, meeting the criteria. With regards to post-development, if I were to revisit this project, I would instead display the game board, and the user interface using a graphical display. Making use of libraries such as tkinter, to produce a simple graphical interface, which would significantly improve the usability of the program. Although, it would take a large amount of time to implement.

At the end of a player's placement, they will be correctly scored, depending on the tiles used, any surrounding tiles, and any bonus tiles, similar to the original game.

```
It's Luc's turn!
-------Your Letters-------
| I | O | Y | E | Z | I |   |
Enter the word to place
or enter "pass!" to forfeit your turn
zone
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
1
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
h8
>Used blank for N
Associate tile scores:  [10, 1, 0, 1]
Existing:  []
Surrounding:  0
Total:  12

   A B C D E F G H I J K L M N O
 1 6 + + 1 + + + 6 + + + 1 + + 6
 2 + 2 + + + 3 + + + 3 + + + 2 +
 3 + + 2 + + + 1 + 1 + + + 2 + +
 4 1 + + 2 + + + 1 + + + 2 + + 1
 5 + + + + 2 + + + + + 2 + + + +
 6 + 3 + + + 3 + + + 3 + + + 3 +
 7 + + 1 + + + 1 + 1 + + + 1 + +
 8 6 + + 1 + + + Z O N E 1 + + 6
 9 + + 1 + + + 1 + 1 + + + 1 + +
10 + 3 + + + 3 + + + 3 + + + 3 +
11 + + + + 2 + + + + + 2 + + + +
12 1 + + 2 + + + 1 + + + 2 + + 1
13 + + 2 + + + 1 + 1 + + + 2 + +
14 + 2 + + + 3 + + + 3 + + + 2 +
15 6 + + 1 + + + 6 + + + 1 + + 6
```

...

```
It's Will's turn!
-------Your Letters-------
| N | U | E | I | W | D | T |
Enter the word to place
or enter "pass!" to forfeit your turn
d nut
Enter "1" for right, "2" for down
or "RE" to reattempt your turn
2
Enter a coordinate in the format: "A1"
or "RE" to reattempt your turn
i7
Associate tile scores:  [2, 0, 1, 1, 1]
>Double letter bonus!
>Double letter bonus!
Existing:  [1]
Surrounding:  0
Total:  9

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + 1 + + 6   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + 3 + + + 2 +   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + + 2 + +   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + + 2 + + 1   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + + 2 + + + + + 2 + + + +   [Y]4  [Z]10 [ ]0
 6 + 3 + + + 3 + + + 3 + + + 3 +   Bonus types:
 7 + + 1 + + + 1 + D + + + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 6 + + 1 + + + Z O N E 1 + + 6   Helpful Tips:
 9 + + 1 + + + 1 + N + + + 1 + +   1. The coordinate is the first letter's
10 + 3 + + + 3 + + U 3 + + + 3 +   position.
11 + + + + 2 + + + T + 2 + + + +   2. When using a placed letter, use a [space]
12 1 + + 2 + + + 1 + + + 2 + + 1   in its place.
13 + + 2 + + + 1 + 1 + + + 2 + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + 3 + + + 2 +   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + + 6
```

I am proud with the method of scoring for this project, as it was fairly comprehensive, in the operation to which scores are applied, for example, the scoring of extra formed words that may be formed during the placement. Also, the specific operation in which bonuses are applied. The documentation during the development of the Scoring sub system, describes the dependencies and process by which this method calculates the final score.
But, to briefly illustrate its usage, I have added some debug messages to show the different sections of scoring that are applied. For example, in the first output, the score is simply equal to the associate tile scores of the word entered, being 10,1,0,1, noting that the blank is successfully scored as zero. With the final score/total being equal to 12, which is correct.

The second example is more complicated as it involves bonuses, also using existing tiles from the board, "O". To begin with it correctly grabs the associate tile scores of the users input, note that the existing tile used is initially scored as zero. Then the function, detects the usage of an existing tile, which is then also scored, as "1". The function also detects the usage of two DLS bonuses, whereby the tile that landed on that bonus is doubled in value. So, the initial tile score of: 2,0,1,1,1 becomes 4,0,2,1,1, the sum of this and the existing tile score is made, giving a total of 9, which is the correct score, earned.

Although this criterion has been fully met, however, the unfortunate limitation with the usage of existing blanks, as mentioned above, means that in the unlikely case that it may occur, a player will be scored incorrectly. Meaning, that this point cannot be fully met. I described a possible solution to this when it was first mentioned above.

At the end of the game the winner is the player with the highest score.

```
>No tiles left in bag to redistribute

   A B C D E F G H I J K L M N O   Letter Scoring:
 1 6 + + 1 + + + 6 + + + P + + M   [A]1  [B]3  [C]3  [D]2  [E]1  [F]4
 2 + 2 + + + 3 + + + Z O O + C O   [G]2  [H]4  [I]1  [J]8  [K]5  [L]1
 3 + + 2 + + + 1 + 1 + + O H + T   [M]3  [N]1  [O]1  [P]3  [Q]10 [R]1
 4 1 + + 2 + + + 1 + + B R I D E   [S]1  [T]1  [U]1  [V]4  [W]4  [X]8
 5 + + + L 2 + + + + + I + S + +   [Y]4  [Z]10 [ ]0
 6 + 3 + U + 3 + + + 3 N O S E +   Bonus types:
 7 + + 1 E + + 1 C 1 + N + 1 + +   TWS:6  DWS:2  TLS:3  DLS:1
 8 R E P S + + Q U I T E 1 + + J   Helpful Tips:
 9 + + O + + + 1 T 1 I D + 1 + I   1. The coordinate is the first letter's
10 H 3 K + B O O T + M + + V I N   position.
11 U + I + L + Y E T I 2 + I + +   2. When using a placed letter, use a [space]
12 G E N R E + + D + D E L V E R   in its place.
13 E + G + W + 1 + 1 + + + I + +   3. When using a blank, enter the letter
14 + 2 + + + 3 + + + F U N D E R   as if you have it.
15 6 + + 1 + + + 6 + + + 1 + W O
Tom has used all of their tiles!

==Game over==
Tom  is the winner!
with  141  points!
>>>
```

Here you can see the final turn output, after a last placement is made, this output is taken from the Final gameplay output. Displaying the message that "Tom has used all of their tiles!", which is one of the two methods that the game can end, the other being by forfeit. After this, the program compares each player's score, then announces the winner, and the amount of points they had.

As much as this fully meets the criterion, it would have been nice to display the ranking, and scoring of the other players, as well. But, due to the limitation in time, and scope, developing a leaderboard system would require more time than was available. Although, this is a feature that would be implemented in post-development, and could be achieved by implementing a sorting algorithm, such as a merge sort.

The game should follow the original Scrabble rules.

This final point combines elements from each system, as well as the majority of the criteria that has been evaluated above. After reviewing all the points, I am confident that this project follows most, if-not-all of the original scrabble rules. Successfully recreating the experience of the Scrabble board game.

Bibliography

Scrabble reference material:

General information regarding gameplay and examples - www.scrabble.hasbro.com/en-us/rules

Tile scoring and distributions – www.en.wikipedia.org/wiki/Scrabble_letter_distributions

Existing works:

Word finders - www.wordplays.com/scrabble-word-finder

Scrabble in C# - https://www.youtube.com/watch?v=pMrdpgNVV4A

Scrabble in Python with tk - https://www.youtube.com/watch?v=A0aZdGYsGJ8

Words with Friends - https://play.google.com/store/apps/details?id=com.zynga.words&hl=en_GB

Luc Heaton