ASSIGNMENT 3

COMP 202, Winter 2020

Due: Tuesday, March 24th 2020, (23:59)

Please read the entire PDF before starting. You must do this assignment individually.

It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment, in some cases through automated tests. While these tests will never be used to determine your entire grade, they speed up the process significantly, which allows the TAs to provide better feedback and not waste time on administrative details. Plus, if the TA is in a good mood while he or she is grading, then that increases the chance of them giving out partial marks. :)

Up to 30% can be removed for bad indentation of your code as well as omitting comments, or poor coding structure.

To get full marks, you must:

- Follow all directions below
  - In particular, make sure that all file names, and function names are **spelled exactly** as described in this document. Otherwise, you might lose all points associated with that module/function.
- Make sure that your code runs.
  - Code with errors will receive a very low mark.
- Write your name and student ID as a comment in all .py files you hand in
- Name your variables appropriately
  - The purpose of each variable should be obvious from the name
- Comment your work
  - A comment every line is not needed, but there should be enough comments to fully understand your program
- Avoid writing repetitive code, but rather call helper functions! You are welcome to add additional functions if you think this can increase the readability of your code.
- Lines of code should NOT require the TA to scroll horizontally to read the whole thing. Vertical spacing is also important when writing code. Separate each block of code (also within a function) with an empty line.

Part 1 (0 points): Warm-up

Do **NOT** *submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that you consult the TAs during their office hours; they can help you and work with you through the warm-up questions. You are responsible for knowing all of the material in these questions.*

Warm-up Question 1 (0 points)

Write a function `get_largest_sublist` which takes as input a list of lists and returns the length of the largest sublist. For example,

```
>>> get_largest_sublist([['cat'], [1, 5, True], [5.3, False]])
3
```

Warm-up Question 2 (0 points)

Write a function `same_elements` which takes as input a two dimensional list and returns true if all the elements in each sublist are the same, false otherwise. For example,

```
>>> same_elements([[1, 1, 1], ['a', 'a'], [6]])
True
>>> same_elements([[1, 6, 1], [6, 6]])
False
```

Warm-up Question 3 (0 points)

Write a function `flatten_list` which takes as input a two dimensional list and returns a one dimensional list containing all the elements of the sublists. For example,

```
>>> flatten_list([[1, 2], [3], ['a', 'b', 'c']])
[1, 2, 3, 'a', 'b', 'c']
>>> flatten_list([[]])
[]
```

Warm-up Question 4 (0 points)

Complete the case study on multidimensional lists presented in class on Tuesday Feb. 25th. You can find the instructions on myCourses.

Warm-up Question 5 (0 points)

Write a function `get_most_valuable_key` which takes as input a dictionary mapping strings to integers. The function returns the key which is mapped to the largest value. For example,

```
>>> get_most_valuable_key({'a' : 3, 'b': 6, 'g': 0, 'q': 9})
'q'
```

Warm-up Question 6 (0 points)

Write a function `add_dicts` which takes as input two dictionaries mapping strings to integers. The function returns a dictionary which is a result of merging the two input dictionary, that is if a key is in both dictionaries then add the two values.

```
>>> d1 = {'a':5, 'b':2, 'd':-1}
>>> d2 = {'a':7, 'b':1, 'c':5}
>>> add_dicts(d1, d2) == {'a': 12, 'b': 3, 'c': 5, 'd': -1}
True
```

Warm-up Question 7 (0 points)

Create a function `reverse_dict` which takes as input a dictionary `d` and returns a dictionary where the values in `d` are now keys mapping to a list containing all the keys in `d` which mapped to them. For example,

```
>>> a = reverse_dict({'a': 3, 'b': 2, 'c': 3, 'd': 5, 'e': 2, 'f': 3})
>>> a == {3 : ['a', 'c', 'f'], 2 : ['b', 'e'], 5 : ['d']}
True
```

Note that the order of the elements in the list might not be the same, and that's ok!

Part 2

The questions in this part of the assignment will be graded.
The main learning objectives for this assignment are:

- Apply what you have learned about list, one dimensional or multidimensional.
- Apply what you have learned about dictionaries.
- Understand how to test functions that return dictionaries or might contain randomness.
- Solidify your understanding of working with loops and strings.
- Create a more complex program which consists of several modules.
- Understand how to write a docstring and use doctest when working with dictionaries.
- Learn to identify when using the function `enumerate` can help you write a cleaner code.

Note that, as always, assignments are designed for you to be practicing what you have learned in class. You can fully complete the assignment using material covered in class up to Tuesday March 10th. You are welcome to include any of the material covered also in later classes if you wish. **You are NOT allowed** to use what we have not seen in class (for example, you cannot use list or dictionary comprehension). You will be heavily penalized if you do so.

For full marks, make sure to add the appropriate documentation string (docstring) to *all* the functions you write. The docstring must contain the following:

- The type contract of the function.
- A brief description of what the function is expected to do.
- At least 3 examples of calls to the function. Make sure to include examples for the different scenarios including edge cases if needed. You are allowed to use *at most* one example per function from this pdf.

Please note that all the examples are given as if you were to call the functions from the shell.

Question 1: Simplified Scrabble (100 points)

Scrabble is a "board-and-tile game in which two to four players compete in forming words with lettered tiles on a 225-square board; words spelled out by letters on the tiles interlock like words in a crossword puzzle."[1]

For this assignment, you will be implementing all the functions needed to build a simplified version of scrabble. Let's first start by setting up some basic terminology:

- The game is played with one **board** and several **tiles**.
  - Tiles represents single letters.
  - In a standard game of scrabble, the board has 225 **squares**.
- Each player has a **rack** on which they keep their tiles.
- Tiles are drawn from a **pool** which at the beginning contains all the tiles in the game (in a standard game of scrabble there are 100 tiles).
- Each letter has a value associated to it which is used to determine the **score** associated to a word.
- When a player places tiles on the board, they'll create a *main word* and some possible *hook words*.
- **Hook words** in scrabble are words which can be formed from another word by adding a single letter at either the beginning or the end.

[1] https://www.britannica.com/sports/Scrabble

- If the words generated by the player are valid, the player receives a number of points equivalent to the score of all the new words they have created. This include the score of the main word as well as any hook word.

If you know how to play scrabble you will notice that the program you obtain at the end of the assignment contains some simplifications such as:

- We do not consider blank tiles.
- We do not play with a board where certain squares amplify the score of the words.
- We do not allow players to challenge other players.
- We do not enforce the fact that a new word must be connected with the cluster of tiles already on the board. A new word can be added anywhere on the board.

Note that, adding the features above would not make the assignment more conceptually difficult, but it would make it much longer. This is why we decided to implement a simplified version of the game.

For this assignment, you are asked to write three modules with several functions which, together with the module provided by us, create a program that allows you to play a simplified version of scrabble on your computers.

The board (30 points)

Let's start by creating a module called `board_utils` which contains several helper functions needed to implements a scrabble board. For full marks, all the following functions must be part of this module:

- `create_board`: This function takes as input two integers representing the number of rows and columns respectively. The function returns a two dimensional list of strings, where all the elements of the sublists are strings containing only the space character. **We indicate with a space character an empty square on the board.** The function should raise a `ValueError` if the inputs are not both positive integers. For example,

```
>>> create_board(3, 3)
[[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
>>> create_board(2, 5)
[[' ', ' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ', ' ']]
>>> create_board(0, 5)
Traceback (most recent call last):
ValueError: Inputs must be positive
```

- `display_board`: This function takes a two dimensional list of strings as input representing a board. From now on, if we say that a list represents a board, you can assume that it is a rectangular list (all sublists are of the same size), that both dimensions are positive, and that all the elements of the sublists are strings with exactly one character. The function displays the board, one row per line, as follows:

```
>>> b = create_board(3, 3)
>>> display_board(b)
    0   1   2
  +-----------+
0 |   |   |   |
  +-----------+
1 |   |   |   |
  +-----------+
2 |   |   |   |
  +-----------+
>>> b = [[' ', ' ', ' ', ' ', ' '], [' ', 'c', 'a', 't', 's']]
>>> display_board(b)
    0   1   2   3   4
  +-------------------+
0 |   |   |   |   |   |
  +-------------------+
1 |   | C | A | T | S |
  +-------------------+
```

To get full marks:

- The numbers indicating the column number must be displayed at the top.
- The numbers indicating the row number must be displayed on the left.
- For each row, the elements of the corresponding sublist must be displayed in the correct order.

It does not matter whether you capitalize the letters or if the board looks exactly like the one above.

- `get_vertical_axis`: This function takes as input a two dimensional list of strings representing a board, and an integer representing the number of a column. The function returns a list of strings containing all the elements from the board on the specified column. For example,

```
>>> b = [['c', 'a', 't', ' '], [' ', 'a', 'r', 't'], [' ', ' ', 'a', ' '], \
[' ', ' ', 'i', ' '], [' ', ' ', 'n', ' ']]
>>> get_vertical_axis(b, 2)
['t', 'r', 'a', 'i', 'n']

>>> get_vertical_axis(b, 0)
['c', ' ', ' ', ' ', ' ']
```

- `find_word`: This function takes as input a list of strings and an integer `i`. It returns the string built by concatenating the sequence of consecutive strings from the list that are not the space characters. This sequence must include the string in position `i`. The function returns the empty string if in position `i` there is a space character. For example,

```
>>> find_word([' ', 'c', 'a', 't', ' '], 1)
'cat'

>>> find_word([' ', 'squi', '', 'rre', 'l'], 2)
'squirrel'

>>> find_word([' ', 'c', 'a', 't', ' ', 'a', 'p', 'p', 'l', 'e'], 7)
'apple'
```

- `available_space`: This function takes as input a list of strings which represents a row/column of the board and an integer `i`. It returns the number of ***empty squares*** on the row/column starting from position `i`. For example,

```
>>> r = ['a', ' ', ' ', 'b', ' ']
>>> available_space(r, 1)
3

>>> r = ['a', ' ', ' ', 'b', ' ', ' ', 'c', ' ', ' ']
>>> available_space(r, 2)
5
```

```
>>> r = ['a', ' ', ' ', 'b', ' ', ' ', 'c', ' d', ' ']
>>> available_space(r, 3)
3
```

- `fit_on_board`: This function takes as input a list of strings representing a row/col on the board, a string `letters`, and an integer `i`. It returns true if the square in position `i` is empty and if there is enough space on the board to fit all the characters in `letters` starting from position `i`, false otherwise. Note that each square on the board can contain at most one letter. This function should **not** modify the input list. For example,

  ```
  >>> a = ['a', ' ', ' ', 'b', ' ']
  >>> fit_on_board(a, 'cat', 1)
  True

  >>> a = ['a', ' ', ' ', 'b', ' ', ' ', 'c', ' ', ' ']
  >>> fit_on_board(a, 'apple', 4)
  False

  >>> a = ['a', ' ', ' ', 'b', ' ', ' ', 'c', ' d', ' ']
  >>> fit_on_board(a, 'pear', 3)
  False
  ```

Tiles and Words (35 points)

Let's now create a module called `dicts_utils`. In this module you will write several helper functions that we'll use to implement the part of scrabble in which we need to validate words and compute their score.

In this module all your functions work with dictionaries. Remember that the order of the keys in a dictionary is not something that Python fixes. This means that if you have a doctest where the output is a dictionary, you could sometime "fail" the doctest even if your code is correct. This might happen simply because the keys are displayed in a different order than you expected. To write doctests that will consistently pass, you can instead test properties of a dictionary. For example:

```
>>> dict = {'d': 1, 'a': 4, 'b': 2}
>>> len(dict)
3
>>> dict['a']
4
>>> dict['b']
2
>>> sum(dict.values())
7
>>> sorted(dict.values())
[1, 2, 4]
>>> dict == {'a': 4, 'b': 2, 'd': 1}
True
```

For full marks, all the following functions must be part of this module:

- `count_occurrences`: This function takes as input a string. It returns a dictionary mapping characters to integers. The keys in the dictionary are the characters from the input string, the values represent the number of occurrences of those characters in the input string. For example,

  ```
  >>> d = count_occurrences('banana')
  >>> d == {'b':1, 'a':3, 'n':2}
  True
  ```

- `flatten_dict`: This function takes as input a dictionary where all the values are non-negative integers. It returns a list containing the keys in the dictionary. Each key should appear in the list as many times as the value associated to such key. The function should **not** modify the input dictionary. For example,

  ```
  >>> d = {'a' : 2, 'f' : 1, 'k' : 5}
  >>> flatten_dict(d)
  ['a', 'a', 'f', 'k', 'k', 'k', 'k', 'k']

  >>> d = {'cat': 2, 'dog': 0, 'bunny' : 3}
  >>> flatten_dict(d)
  ['cat', 'cat', 'bunny', 'bunny', 'bunny']
  ```

  Note that the order of the elements in the list does not matter. It is possible that your code is correct, but you still fail the doctest above. To write a doctest that will consistently pass, you should instead test properties about the list. For example,

  ```
  >>> d = {'cat': 2, 'dog': 0, 'bunny' : 3}
  >>> animals = flatten_dict(d)
  >>> animals.count('cat')
  2
  >>> animals.count('bunny')
  3
  >>> animals.count('dog')
  0
  >>> animals.sort()
  >>> animals
  ['bunny', 'bunny', 'bunny', 'cat', 'cat']
  ```

- `get_word_score`: This function takes as input a string and a dictionary mapping characters to integers representing the number of points each character is worth. If a given character is not a key in the input dictionary, then you should assume that such character is worth 0 points. The function returns the score of a word (the input string) computed by summing together the value of each character in the word. For example,

  ```
  >>> v = {'a': 5, 't': 3, 'n': -2}
  >>> get_word_score('cat', v)
  8

  >>> get_word_score('banana', v)
  11

  >>> v = {}
  >>> get_word_score('banana', v)
  0
  ```

- `is_subset`: This function takes as input two dictionaries where all the values are non-negative integers. It returns true if the first dictionary can be considered to be a subset of the second one. We consider a dictionary `d` to be a subset of another dictionary `b` if all the keys in `d` are keys in `b` and the value associated to each key in `d` is smaller than or equal to the value associated to the same key in `b`. The function should **not** modify the input dictionaries. For example,

  ```
  >>> a = {'a': 2, 'c': 1}
  >>> b = {'a': 2, 'b': 1, 'c': 2}
  >>> c = {'a': 1, 'c': 3}
  ```

```
>>> is_subset(a, b)
True

>>> is_subset(b, a)
False

>>> is_subset(a, c)
False
```

- `subtract_dicts`: This function takes as input two dictionaries `d1` and `d2` where all the values are non-negative integers. If `d2` is a subset of `d1`, then the function updates `d1` by replacing the values associated to the common keys with the difference between the original value in `d1` and the value in `d2`. Otherwise, `d1` remains as is. The function returns true if `d2` was a subset of `d1`, false otherwise. Note that the function should **not** modify `d2`. For example,

```
>>> a = {'a': 2, 'c': 1}
>>> b = {'a': 2, 'b': 1, 'c': 2}
>>> c = {'a': 5, 'b': 3}

>>> subtract_dicts(b, a)
True
>>> b == {'b': 1, 'c': 1}
True

>>> subtract_dicts(c, a)
False
>>> c == {'a': 5, 'b': 3}
True
```

- `create_scrabble_dict`: This function takes as input a list of strings. It returns a dictionary that maps integers representing the number of characters in a word to a dictionary of words with the specified length. The latter maps a single letter to a list of words beginning with such letter. For example,

```
>>> w = ['aa', 'qi', 'za', 'cat', 'can', 'cow', 'dog', 'dad', 'hippo', 'umami', 'uncle']
>>> d = create_scrabble_dict(w)
>>> d == {2 : {'a': ['aa'], 'q': ['qi'], 'z': ['za']}, 3 : {'c': ['cat', 'can', 'cow'], \
'd': ['dog', 'dad']}, 5 : {'h': ['hippo'], 'u' : ['umami', 'uncle'] }}
True
```

- `is_valid_word`: This function takes as input a string and a dictionary. The dictionary has the same format as the one returned by the function `create_scrabble_dict`. This function returns true if the input string appears in the dictionary, false otherwise. For example,

```
>>> w = ['aa', 'qi', 'za', 'cat', 'can', 'cow', 'dog', 'dad', 'hippo', 'umami', 'uncle']
>>> d = create_scrabble_dict(w)
>>> is_valid_word('hippo', d)
True

>>> is_valid_word('zebra', d)
False

>>> is_valid_word('pear', d)
False
```

Scrabble helpers (35 points)

Finally, let's create a module called `scrabble_utils`. In this module you will write several functions that we'll use to implement the general mechanics of a game of scrabble.

We will represent a player's **rack** with a dictionary mapping single letter strings to non-negative integers. Let `r` be such dictionary, and `c` be a string containing one single letter. Then `r[c]` denotes how many **tiles** with letter `c` are on the rack. We'll use a similar dictionary also to represent the **pool** from which players draw their tiles.

For full marks, all the following functions must be part of this module:

- `display_rack`: This function takes as input a dictionary representing the rack of a player (see description above). The function displays one line containing the letters that are on the rack using upper case. For example,

  ```
  >>> display_rack({'a': 2, 'f': 1, 'g': 2, 'o': 1, 'z': 1})
  A A F G G O Z

  >>>>>> display_rack({'g': 2, 'k': 0, 'p': 4})
  G G P P P P
  ```

- `has_letters`: This function takes as input a dictionary representing the rack of a player (see description above), and a string. The function returns true if all the characters in the input string are available on the rack and if so, it removes those letters from the rack. Otherwise, the function returns false and does not modify the rack. For example,

  ```
  >>> r = {'a': 2, 'c': 1, 't': 1, 'i': 2, 'r': 1}
  >>> has_letters(r, 'cat')
  True
  >>> r == {'a': 1, 'i': 2, 'r': 1}
  True

  >>> r = {'a': 2, 'c': 1, 't': 1, 'i': 2, 'r': 1}
  >>> has_letters(r, 'tiara')
  True
  >>> r == {'i': 1, 'c': 1}
  True

  >>> r = {'a': 2, 'c': 1, 't': 1, 'i': 2, 'r': 1}
  >>> has_letters(r, 'track')
  False
  >>> r == {'a': 2, 'c': 1, 't': 1, 'i': 2, 'r': 1}
  True

  >>> has_letters(r, 'tract')
  False
  >>> r == {'a': 2, 'c': 1, 't': 1, 'i': 2, 'r': 1}
  True
  ```

- `refill_rack`: This function takes as input two dictionaries (one representing the rack of a player, the other representing the pool of letters, respectively) and a positive integer `n`. The function draws letters at random from the pool and adds them to the rack until there are either `n` letters on the rack or no more letters in the pool. The function does not return anything and it might modify both input dictionaries. For example,

```
>>> random.seed(5)
>>> r1 = {'a': 2, 'k': 1}
>>> b = {'a': 1, 'e': 2, 'h': 1, 'l': 2, 'n': 1, 'p': 2, 's': 3, 't': 2, 'z': 1}
>>> refill_rack(r1, b, 7)
>>> r1
{'a': 2, 'k': 1, 's': 1, 'l': 1, 't': 1, 'n': 1}
>>> b
{'a': 1, 'e': 2, 'h': 1, 'l': 1, 'p': 2, 's': 2, 't': 1, 'z': 1}

>>> r2 = {'e': 3, 'q' : 1, 'r' : 1}
>>> refill_rack(r2, b, 8)
>>> r2
{'e': 3, 'q': 1, 'r': 1, 'z': 1, 's': 1, 'a': 1}
>>> b
{'e': 2, 'h': 1, 'l': 1, 'p': 2, 's': 1, 't': 1}

>>> refill_rack(r2, b, 5)
>>> r2
{'e': 3, 'q': 1, 'r': 1, 'z': 1, 's': 1, 'a': 1}
>>> b
{'e': 2, 'h': 1, 'l': 1, 'p': 2, 's': 1, 't': 1}
```

Note that you can use the function `random.choice` to selected a random element out of a list. Note also that if in the pool there are five a's and one b, then the probability of drawing an a should be 5 times larger than the probability of drawing the b.

- `compute_score`: This function takes as input a list of strings, a dictionary mapping letters to integers representing the number of points each letter is worth, and a dictionary representing valid words (with the same format as the one returned by the function `dict_utils.create_scrabble_dict`). The function returns the score obtained by summing together the score of each word from the input list. If any of the words in the list is not valid, then the total score should be 0. For example,

```
>>> v = {'a': 1, 'p': 3, 'h': 2}
>>> w = ['aa', 'qi', 'za', 'cat', 'can', 'cow', 'dog', 'dad', 'hippo', 'umami', 'uncle']
>>> d = dicts_utils.create_scrabble_dict(w)

>>> compute_score(['hippo', 'aa'], v, d)
10

>>> compute_score(['umami', 'zebra'], v, d)
0

>>> compute_score(['qi'], v, d)
0
```

- *NOTE: this function is the most logically difficult in the entire assignment. Do not feel discouraged if you do not know how to tackle it right away. Make sure to follow the discussion board on Piazza and add any questions you might have on the board. We suggest you try to first figure out how to solve this task on paper. Try to break down every single action you take (as a human) to perform this task. Then think about how to translate these actions into code.*

  `place_tiles`: This function takes as input a two dimensional list representing the board, a string representing the letters the player wants to add to the board, two integers representing the row and the column number (respectively) of the starting square, and finally a string indicating the direction to take when placing the letters on the board (either ‘down’ or ‘right’). The function adds the letters received as input to the board given a starting position, and a direction. It returns a list of words created by adding those letters to the board. This list will contain the main word as well as any hook word generated. The order of the elements in this list does not matter. Note that the function should modify the input list, unless the direction provided is not equal to neither ‘down’ nor ‘right’. In such case, the function returns an empty list. **In this function, you can assume that the starting square is empty and that the provided letters will fit on the board.**

Finally, note that it very tedious to write docstest using `display_board` since they might fail for very small details (one space too many or too few). We suggest you write the doctest using directly the list. Below we opted for writing the examples displaying the board because the result is more clear visually.

Here are some examples of using this function:

```
>>> b = [[' ', ' ', ' ', ' '], [' ', ' ', ' ', ' '], [' ', ' ', ' ', ' '], \
[' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ']]

>>> place_tiles(b, 'cat', 0, 0, 'right')
['cat']
>>> board_utils.display_board(b)
      0   1   2   3
    +---------------+
0   | C | A | T |   |
    +---------------+
1   |   |   |   |   |
    +---------------+
2   |   |   |   |   |
    +---------------+
3   |   |   |   |   |
    +---------------+
4   |   |   |   |   |
    +---------------+


>>> place_tiles(b, 'rain', 1, 2, 'down')
['train']
>>> board_utils.display_board(b)
    0   1   2   3
  +---------------+
0 | C | A | T |   |
  +---------------+
1 |   |   | R |   |
  +---------------+
2 |   |   | A |   |
  +---------------+
3 |   |   | I |   |
  +---------------+
4 |   |   | N |   |
  +---------------+

>>> words = place_tiles(b, 'at', 1, 1, 'right')
>>> words.sort()
>>> words
['aa', 'art']
```

```
>>> board_utils.display_board(b)
    0   1   2   3
  +---------------+
0 | C | A | T |   |
  +---------------+
1 |   | A | R | T |
  +---------------+
2 |   |   | A |   |
  +---------------+
3 |   |   | I |   |
  +---------------+
4 |   |   | N |   |
  +---------------+
```

- `make_a_move`: This function takes as input a list representing the board, a dictionary representing the player's rack, a string representing the letters the player wants to place, two integers representing the row and the column number (respectively) of the starting square on the board, and a string representing a direction (either 'down' or 'right'). If the direction received is neither 'down' nor 'right', the function terminates right away by returning an empty list. Otherwise, the function checks if this is a valid move: is there enough space on the board to place those letters? Does the player actually have those letters on their rack? If so, then the letters are place on the board and a list of words created by performing the move is returned. Otherwise, if the letters do not fit on the board, the function raises an `IndexError`. If they fit, but the player does not have those letters on their rack, then the function raises a `ValueError`. If an error is raised, then neither the board nor the rack is modified. Otherwise, the letters will be removed from the rack and placed on the board. For example,

```
>>> b = [['c', 'a', 't', ' '], [' ', ' ', ' ', ' '], [' ', ' ', ' ', ' '],\
[' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ']]
>>> r = {'a': 3, 't': 2, 'c' : 1, 'r' : 1, 'i' : 1, 'n' : 1}


>>> make_a_move(b, r, 'rain', 1, 2, 'down')
['train']
>>> r == {'a': 2, 't' : 2, 'c' : 1}
True
>>> board_utils.display_board(b)
    0   1   2   3
  +---------------+
0 | C | A | T |   |
  +---------------+
1 |   |   | R |   |
  +---------------+
2 |   |   | A |   |
  +---------------+
3 |   |   | I |   |
  +---------------+
4 |   |   | N |   |
  +---------------+


>>> words = make_a_move(b, r, 'at', 1, 1, 'right')
>>> words.sort()
>>> words
```

```
['aa', 'art']
>>> r == {'a': 1, 't' : 1, 'c' : 1}
True
>>> board_utils.display_board(b)
    0   1   2   3
  +---------------+
0 | C | A | T |   |
  +---------------+
1 |   | A | R | T |
  +---------------+
2 |   |   | A |   |
  +---------------+
3 |   |   | I |   |
  +---------------+
4 |   |   | N |   |
  +---------------+


>>> make_a_move(b, r, 'cats', 0, 2, 'right')
Traceback (most recent call last):
IndexError: Not enough space on the board.
>>> r == {'a': 1, 't' : 1, 'c' : 1}
True
>>> board_utils.display_board(b)
    0   1   2   3
  +---------------+
0 | C | A | T |   |
  +---------------+
1 |   | A | R | T |
  +---------------+
2 |   |   | A |   |
  +---------------+
3 |   |   | I |   |
  +---------------+
4 |   |   | N |   |
  +---------------+
```

Scrabble

With this pdf you can also find a text file (*words.txt*) and Python file (*scrabble.py*). In the module provided we use the functions you have created, together with the text file provided to run a simplified version of scrabble. Once you have finished the assignment, you can use *scrabble.py* to test your functions and play as many games as you like. Make sure to save both the text file and the python file in the same folder as all the others modules you have created. Note that you could have written the code provided on your own. The reason why we are providing this module is just to keep the assignment at reasonable length. **Disclaimer**: the text file contains all valid scrabble words that have no more than 6 characters. This means that if you end up creating a word which contains more than 6 character, the program will not give you any points for it because it will not recognize it as valid. Feel free to add more words to the text file if you'd like, or to use a different text file!

What To Submit

You must submit all your files on codePost (https://codepost.io/). The file you should submit are listed below. Any deviation from these requirements may lead to lost marks.

`board_utils.py`
`dicts_utils.py`
`scrabble_utils.py`
`README.txt` In this file, you can tell the TA about any issues you ran into doing this assignment. If you point out an error that you know occurs in your program, it may lead the TA to give you more partial credit.
This file is also where you should make note of anybody you talked to about the assignment. Remember this is an individual assignment, but you can talk to other students using the **Gilligan's Island Rule**: you can't take any notes/writing/code out of the discussion, and afterwards you must do something inane like watch television for at least 30 minutes.
If you didn't talk to anybody nor have anything you want to tell the TA, just say "nothing to report" in the file.