Word Search

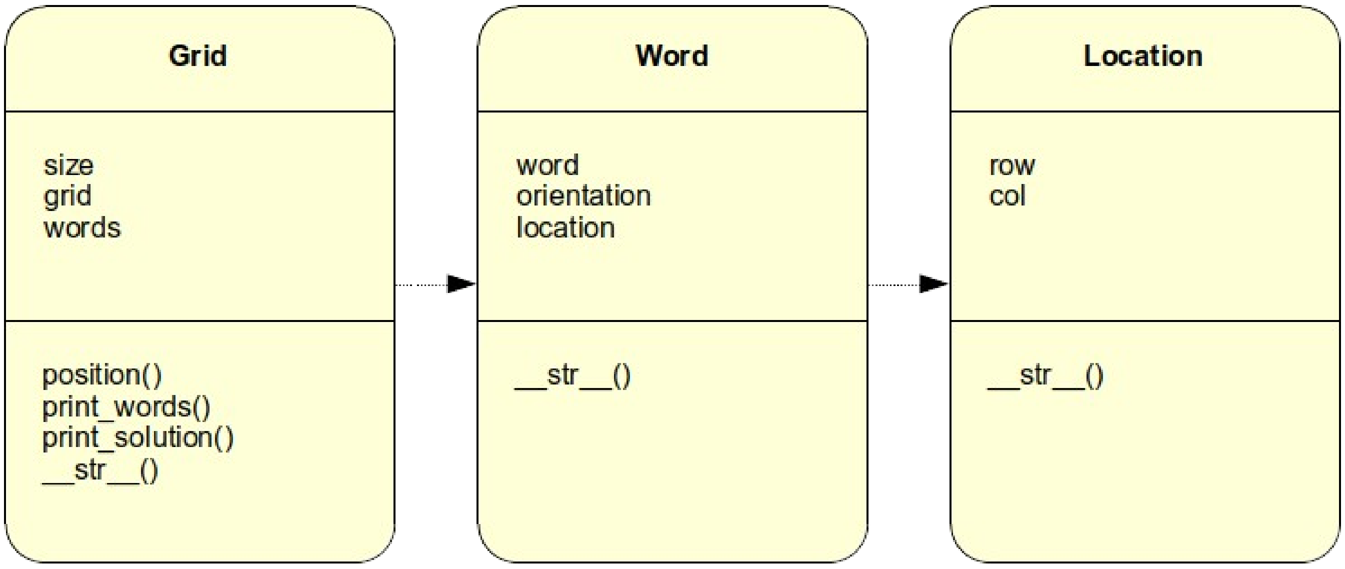
Part 2

Overview

The purpose of this programming assignment is to gain more experience writing object-oriented programs. You will do this by completing the **word search** puzzle that you began in the last programming assignment.

Specification

First, recall the class diagram from the previous programming assignment:



In the last programming assignment, you implemented the *`Word`* and *`Location`* classes. A test program (WordSearch.py) was used to test your implementation of the two classes. In this programming assignment (which is part 2), you will focus on the *`Grid`* class and the main program (we will call this one WordSearch.py again) that drives the process of creating the word search.

Similar to the last programming assignment, the main program will take a list of words as input (via *stdin*), one word per line, and randomly select a specified number of the words. Each selected word will then be positioned in the *`Grid`* (as long as it fits within the *`Grid`*). Finally, the *`Grid`* and *`Word`*s will be displayed. Optionally, the solution to the word search will be displayed. The solution is just the grid without random letters placed in positions that do not already have a letter of one or more of the selected words. More details later.

For this assignment, you will be provided some source code and text files as follows:

- `animals.txt`: a text file containing animal names, one per line (these represent one set of words for input).
- `words.txt`: a (larger) text file containing words, one per line (these represent another set of words for input).
- `WordSearch.py`: a template for the main part of the program that makes use of the *`Location`*, *`Word`*, and *`Grid`* classes.
- `Grid.py`: a **partially implemented** *`Grid`* class that is missing getters and setters, only supports the HR (horizontally and to the right) orientation, and does not sort the *`Word`*s prior to displaying them.
- `Debug.py`: the definition of a *`DEBUG`* Boolean variable that can be used within the entire program to provide meaningful information.

Moreover, you will make use of the following source code that you developed in the last programming assignment:

- `Location.py`: your implementation of the *`Location`* class (as described in the class diagram above).
- `Word.py`: your implementation of the *`Word`* class (as described in the class diagram above).

For this assignment, you will modify Grid.py and WordSearch.py. Although they are provided to you, they are limited and require further development. Furthermore, you will add a small feature to the *`Word`* class that will permit *`Word`* instances to be sorted according to the spelling of words.

Orientations

Although these were discussed in the last programming assignment, we will cover them again here:

- *HR*: horizontally to the right (i.e., from left-to-right). In this orientation, the word is on a single row and is spelled from left-to-right.
- *HL*: horizontally to the left (i.e., from right-to-left). In this orientation, the word is on a single row and is spelled from right-to-left.

- *VD*: vertically down (i.e., from top-to-bottom). In this orientation, the word is on a single column and is spelled from top-to-bottom.
- *VU*: vertically up (i.e., from bottom-to-top). In this orientation, the word is on a single column and is spelled from bottom-to-top.
- *DRD*: diagonally to the right and down (i.e., from top-left to bottom-right). In this orientation, the word is across multiple rows and columns and is spelled from top-left to bottom-right.
- *DRU*: diagonally to the right and up (i.e., from bottom-left to top-right). In this orientation, the word is across multiple rows and columns and is spelled from bottom-left to top-right.
- *DLD*: diagonally to the left and down (i.e., from top-right to bottom-left). In this orientation, the word is across multiple rows and columns and is spelled from top-right to bottom-left.
- *DLU*: diagonally to the left and up (i.e., from bottom-right to top-left). In this orientation, the word is across multiple rows and columns and is spelled from bottom-right to top-left.

The orientations should be stored in a Python list through a **class variable** called *`ORIENTATIONS`* that is located in your *`Word`* class.

The *`Grid`* class

For the *`Grid`* class, you must implement the following:

- The constructor which takes a grid *`size`* as a parameter. Note that the default value for *`size`* should be 25. The constructor should appropriately set the instance variable *`_size`* based on this parameter. Moreover, the constructor should initialize the grid (*`_grid`*) with empty rows and columns as specified by *`size`* and the list of words (*`_words`*).
- Appropriate getters for the instance variables *`_size`*, *`_grid`*, and *`_words`*, and a setter for the instance variable *`_size`*. Note that values less than 1 specified through the setter should default to 25.
- In the *`position()`* function, modify the portion of code that appropriately sets *`min_row`*, *`max_row`*, *`min_col`*, and *`max_col`* to support the remaining orientations (i.e., in addition to the provided HR orientation).
- In the *`_check()`* function, modify the portion of code that changes the current *`row`* and *`col`* to support the remaining orientations (i.e., in addition to the provided HR orientation).

- In the *`_position()`* function, modify the portion of code that changes the current *`row`* and *`col`* to support the remaining orientations (i.e., in addition to the provided HR orientation).
- In the *`print_words()`* function, sort the list of words prior to displaying them.

The *`Word`* class

You will use the *`Word`* class that you designed as the solution for part 1 of this programming assignment. However, you must add the following:

- Override the **less than** operator to return if one word is "less than" another (i.e., it comes before it alphabetically). You will need to implement the *`__lt__()`* function.

This additional feature will permit you to sort the list of words in your *`Grid`* class.

The *WordSearch* main program

This class implements the word search and creates the puzzle. It then displays the puzzle, the words to find within it, and the solution (if specified). Sample output is provided below.

A template for this class will be provided to you and will serve as a starting point. Note that, as you implement this class, you may need to import additional libraries. Comments are included throughout the template that assist you in implementing its required features as follows:

- Initialize a list of words that will be read from *stdin*.
- Read the words, one word at a time, into the initialized list. Ensure that trailing newlines are removed and that the words are converted to uppercase.
- Randomly select a sample of *`NUM_WORDS`* words from the list. Note that the constant *`NUM_WORDS`* is declared at the top of the template.
- Initialize the grid as an instance of the *`Grid`* class.
- For each word in the list of words:
  - Randomly select an orientation from those defined in the class variable *`ORIENTATIONS`* in the *`Word`* class.
  - Using the *`position()`* function in the *`Grid`* class, try to position the current word in the grid at the specified orientation.
- Display statistics of the word search puzzle in the following form: *Successfully placed X of Y words*. The *`Grid`* function tries to place each word in the grid up to *`MAX_TRIES`* times. The class variable *`MAX_TRIES`* is defined in the *`Grid`* class. The total number of words placed in the grid is the length of the instance variable *`_words`* in the *`Grid`*

class. The total number of words selected from the input is the length of the variable `words` in this template.

- Display the grid (i.e., the grid with random letters placed in positions that do not already have letters from the words successfully placed in the grid).
- Display the words successfully placed in the grid (beneath the grid).
- Display the solution if specified (i.e., if the constant `DISPLAY_SOLUTION` at the top of the template is set to *true*).

Constraints and notes

Note the following constraints and notes (see the rubric below for more detail):

- You must submit a ZIP file containing Grid.py, WordSearch.py, your updated Word.py, and your (likely unmodified) Location.py. These filenames are required. The class names `Grid` and `Word` are also required. To be clear, do not change the names of the classes and filenames. This should be evident based on the provided source code in WordSearch.py.
- Your output must be similar to the output of the sample runs shown below for the specified user inputs. Since the words are randomly chosen, then they will likely be different.
- You must not integrate the contents of others files (i.e., Debug.py, WordSearch.py, animals.txt, and/or words.txt) into the source code where you implement the required classes. For example, do not add the `DEBUG` variable to Word.py or to Grid.py.
- You must comment your source code appropriately, include an informative header at the top of each file, and use good coding style.
- You must use meaningful variable names.
- Input must be from *stdin*.
- You must use standard naming conventions for the instance variables (e.g., `_row`, `_col`, `_orientation`, etc).
- You must not modify the following provided files: Debug.py, animals.txt, and words.txt. Obviously, you will modify Grid.py and WordSearch.py as specified above.
- As before, the orientations must be declared as a class variable called `ORIENTATIONS` in the `Word` class. Furthermore, `ORIENTATIONS` must be a list of the strings defined above.
- In the `Word` class, you must implement a comparison of words to support sorting them (which is done in the `Grid` class). To do this, you will need to overload the **less than** operator via the `__lt__()` function.

- In the *`Grid`* class, the constructor should take *`size`* as a parameter. Moreover, you must specify a default parameter value for *`size`* (25). The constructor should set the appropriate value for the instance variable *`_size`*, properly initialize *`_grid`*, and properly initialize *`_words`*.
- In the *`Grid`* class, you must implement getters for the instance variables *`_size`*, *`_grid`*, and *`_words`*; and a setter for the instance variable *`_size`*.
- In the *`Grid`* class, you must implement range checking in the setter for the instance variable *`_size`* such that it is greater than 0. If a value less than 1 is provided, it should default to 25.
- In the *`Grid`* class, you must add support for the remaining orientations (other than the provided HR orientation). This includes appropriately setting *`min_row`*, *`max_row`*, *`min_col`*, and *`max_col`* in the *`position()`* function; *`row`* and *`col`* in the *`_check()`* function; and *`row`* and *`col`* in the *`_position()`* function.
- In the *`Grid`* class, you must sort the list of words prior to displaying them in the *`print_words()`* function using the *`list.sort()`* function.

Deliverable

Submit a ZIP file containing the **four** Python 3 source files (Location.py (even if unmodified from part 1 of the programming assignment), Word.py, Grid.py, and WordSearch.py) representing the three classes (*`Location`*, *`Word`*, and *`Grid`*) and the main program (WordSearch.py). They will be combined with the provided source file (Debug.py) and input files (animals.txt and words.txt), and be executed via the command line (e.g., `python3 WordSearch.py < animals.txt`).

Sample output

Here is some sample output of various runs of a **correct** solution:

Sample 1 (*DEBUG* is False, *NUM_WORDS* is 5, *GRID_SIZE* is 10, *DISPLAY_SOLUTION* is False)

```
Successfully placed 5 of 5 words.

K B H Y K J Z C L S
K R X C V J J Z O J
D H A I F T S R V L
```

```
I O K H Y Y E P M I
Q T L M S C T A G Z
X R W P O X E N P A
X K Y N H G J T I R
K R I I Y I U H J D
E H C R X M N E X F
R S M B C M G R N Y

DOLPHIN
LIZARD
PANTHER
RHINOCEROS
SHARK
```

Sample 2 (*`DEBUG`* is False, *`NUM_WORDS`* is 5, *`GRID_SIZE`* is 10, *`DISPLAY_SOLUTION`* is True)

```
Successfully placed 4 of 5 words.

A L L I G A T O R T
F H Y H A T E E H C
M A Z D C S Y P Q V
G B I R N N P W J F
F A M Z H M N X N S
U R I V O D U J E E
B C D N H F A X Y A
Q I K U U X F D Y A
F E K E N X Y A Y Q
Y N P Q V B N D H C




ALLIGATOR
CHEETAH
CRAB
MONKEY

A L L I G A T O R .
```

```
. . . H A T E E H C
. . . . . . . . . .
. B . . . . . . . .
. A . . . M . . . .
. R . . O . . . . .
. C . N . . . . . .
. . K . . . . . . .
. E . . . . . . . .
Y . . . . . . . . .
```

Note that the words are randomly chosen and will therefore be different for every run.

Hints

Here are some hints to help get you started:

- Of course, feel free to Google things to help you. As always, do not Google solutions and make sure to **cite sources**.
- Strive for efficiency. If you implement range checking (e.g., to ensure that *`_size`* in the *`Grid`* class is greater than 0) in the setter, use it (the setter) when initializing the value in the constructor (when instantiating a new object).
- Be precise in your library imports. For example, only import *`DEBUG`* where needed. In this case, it is only in the *`Word`* class.
- When overriding the **less than** operator in the *`Word`* class, make use of the < operator that is already defined for strings.
- In the *`Grid`* class, to check if a specific orientation is within a set of orientations, you can use the *in* membership operator; for example, in the following statement:

  -
    ```
    if (orientation in [ "HR", "DRD", "DRU" ]):
        ...
    ```
- In the *`Grid`* class, strive for efficiency. The provided comments should help with this and hint at how to successfully address the requirements.
- In the WordSearch (main program), strive for efficiency. Similarly, the provided comments should help with this and hint at how to successfully address the requirements.

Rubric

Please note the following rubric for this programming assignment:

Word Search (part 2)		
#	**Item**	**Points**
1	ZIP file submitted and contains the required files	2
2	Good coding style	2
3	Appropriate comments	2
4	Appropriate header	2
5	Meaningful variable names	2
6	Input is from stdin	2
7	Output is correct and properly responds to DISPLAY_SOLUTION	2
8	Word class properly sorts words via __lt__()	2
9	Orientations are properly defined through a class variable in Word	2
10	Grid constructor properly instantiates a Grid	6
11	Grid constructor correctly uses default parameters	2
12	Grid getters are properly included	2
13	Grid setter is properly included with range checking	2
14	Support for the remaining orientations is correctly implemented in Grid position	6
15	Support for the remaining orientations is correctly implemented in Grid __check	4
16	Support for the remaining orientations is correctly implemented in Grid __position	4
17	Grid class properly sorts the words	4
18	Proper naming convention for instance variables	2
TOTAL		**50**