Word Search

Part 1

Overview

The purpose of this programming assignment is to gain experience writing object-oriented programs. You will do this by building the foundation for a word search puzzle.

Word search

A **word search** is basically a puzzle in the form of a game. It consists of words that are hidden within a grid of letters. The words can be hidden in any orientation: horizontally, vertically, and diagonally. Moreover, they can be hidden in any direction (e.g., horizontally from left-to-right, horizontally from right-to-left, vertically from top-to-bottom, vertically from bottom-to-top, diagonally from top-left to bottom-right, and so on). Here is a simple example of a word search that consists of five words hidden in a 10x10 grid (the words are highlighted to make them easier to find):

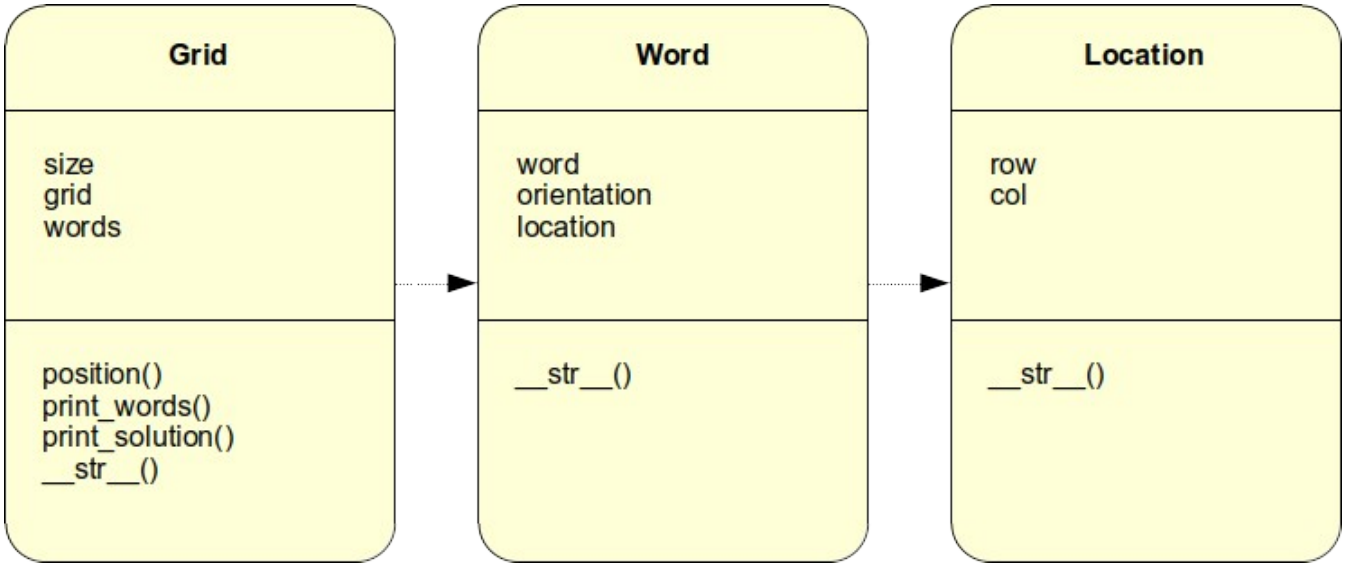
```
U C Y Z G A D D P E
W M E K A N S Z N L
K Q Q J Y L Y P W I
L Z S A A X A E O D
I A E M E N Q W F O
P D F B T A U K S C
G Q U H R K S B K O
U C E I H A S O S R
C R D I G J N P X C
H A L L I G A T O R

PANTHER     ZEBRA
CROCODILE   SNAKE
ALLIGATOR
```

Of course, the number of words can vary (and typically depends on the size of the grid). Moreover, the size of the grid can vary. For the purposes of this programming assignment, the grid will always be as wide as it is tall; that is, the number of letters in each row of the grid will be the same as the number of letters in each column of the grid (i.e., it is a square grid).

Specification

The entire word search game will be completed over two programming assignments. In this first programming assignment, you will lay the foundation for the word search game. In the next programming assignment, you will complete the game.

So how do we begin? Let's define a few classes to help lay the foundation for the game. Although you probably have not yet encountered class diagrams (if not, you will soon), they help to visually identify the classes involved in a program, along with their members; that is, their state (through instance variables) and behavior (through methods). Take a look at the following class diagram:



Each rectangle represents a class. In our word search, there are three classes: `Grid`, `Word`, and `Location`. Think of it like this: the `Grid` has some number of `Word`s hidden within it. Each `Word` is hidden at some `Location` (in the `Grid`).

The class diagram not only identifies the classes, but also adds some details; specifically, the instance variables and the methods within the classes. The `Grid` class contains three instance variables: `size`, `grid`, and `words`. Instance variables represent a **has-a** behavior; therefore, a `Grid` has a size, has a grid (that is made up of rows and columns of letters), and has a list of words (specifically, instances of the `Word` class). The `Grid` class can also position a word into the grid through the method `position()`. It can also print the list of words that are hidden in the grid through the method `print_words()`, and print the solution (i.e., print the hidden words only without the rest of the letters in the grid), through the method

print_solution(). Finally, the method *__str__()* just prints the grid (with all the letters in it).

Similarly, a *Word* has a *word* (the spelling of the word), an *orientation* (e.g., horizontally from left-to-right, diagonally from bottom-right to top-left, etc), and a *location* (which is an instance of the *Location* class). It also has the method *__str__()* which displays a word (in uppercase).

Finally, the *Location* class has a *row* and *col*, identifying a position where the word begins within the *Grid*. It also has the method *__str__()* which displays a location in the format (row,col).

The dashed/dotted arrows in the class diagram (i.e., from *Grid* to *Word* and from *Word* to *Location*) also imply a **has-a** relationship. That is, a *Grid* has a *Word* (well, likely more than just one), and a *Word* has a *Location*. Sometimes we say that one class makes use of another class. For example, *Grid* makes use of *Word*, and *Word* makes use of *Location*. More on that later.

In this programming assignment (which is part 1), you will only focus on the *Word* and *Location* classes. The *Grid* class will be the focus of the next programming assignment. For now, you will write a program that takes a list of words as input (via *stdin*), one word per line, converting each to uppercase. You will then randomly select a specified number of the words. For each selected word, you will randomly select an orientation from the list of orientations (more details will be provided later) and a *Location* (within the size of the currently fictitious grid). You will then create a *Word* instance for each selected word, ensuring that the chosen orientation and *Location* is correctly specified. Finally, you will display the *Word*s.

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`: the main part of the program that makes use of the *Location* and *Word* classes.
- `Debug.py`: the definition of a *DEBUG* Boolean variable that can be used within the entire program to provide meaningful information (more details will be provided later).

For this assignment, you will create the following source code (i.e., these are **not** provided to you):

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

Orientations

The word search will support the following orientations:

- *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 the *`Word`* class. Ideally, declare and initialize the class variable at the beginning of the *`Word`* class before the constructor.

The `Location` class

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

- The constructor which takes a *`row`* and *`col`* as parameters. Note that the default value for *`row`* should be 0, and the default value for *`col`* should be 0. The constructor should appropriately set the instance variables based on these parameters.

- Appropriate getters and setters for the instance variables *_row* and *_col*. Note that, if negative values are specified through the setters, they should default to 0.
- A *__str__()* method that returns a string representation of the *Location* in the format (row,col).

To test your *Location* class, you could implement the following main program:

```
from Location import Location

l1 = Location()
l2 = Location(10, 10)
l3 = Location(-100, 100)
l4 = Location(50, -50)
print(l1)
print(l2)
print(l3)
print(l4)

l1.row = 25
l1.col = 25
l2.row = -10
l2.col = -100
print(l1)
print(l2)
```

Note that, for this example, the test program has been saved to a file called `LocationTest.py` and was executed via `python3 LocationTest.py`. The output of the test program is as follows:

```
(0,0)
(10,10)
(0,100)
(50,0)
(25,25)
(0,0)
```

Pay attention to boundary conditions. Notice how the setters have appropriately handled negative values for *row* and *col*. This is known as **range checking**. Also notice how the constructor defaults *row* and *col* to 0.

The *Word* class

For the *Word* class, you must implement the following:

- The constructor which takes a *word*, an *orientation*, and a *location* as parameters. Note that the default value for *orientation* should be *None*, and the default value for *location* should be *None*. The constructor should appropriately set the instance variables based on these parameters.
- Appropriate getters and setters for the instance variables *_word*, *_orientation*, and *_location*. Note that *_location* should be an instance of the *Location* class.
- A *__str__()* method that returns a string representation of the *Word* in two formats:
  - If *DEBUG* is *false*: just the word in uppercase.
  - If *DEBUG* is *true*: the word, its orientation, and its location in the format word/orientation@location. For example: ZEBRA/HR@(3,5).

Note that your *Word* class will need to import Debug.py to appropriately utilize the *DEBUG* variable. For example:

```
from Debug import DEBUG

...

if (DEBUG):
     ...
```

To test your *Word* class, you could implement the following main program:

```
from Word import Word
from Location import Location

l1 = Location(3, 5)
w1 = Word("zebra", "HR", l1)
l2 = Location(-10, 10)
w2 = Word("Panther", "DLD", l2)
l3 = Location()
w3 = Word("GIRAFFE", "DRU", l3)
print(w1)
print(w2)
print(w3)
```

Note that, for this example, the test program has been saved to a file called `WordTest.py` and was executed via `python3 WordTest.py`. The output of the test program (assuming Debug.py has the statement `DEBUG = False`) is as follows:

```
ZEBRA
PANTHER
GIRAFFE
```

If *DEBUG* is *true*, then the output is as follows:

```
ZEBRA/HR@(3,5)
PANTHER/DLD@(0,10)
GIRAFFE/DRU@(0,0)
```

Note how the words are output in uppercase regardless of how they were specified when creating new instances of the Word class.

The *WordSearch* class

This class will be provided to you and will test your implementation of the *Location* and *Word* classes. When running WordSearch.py, make sure that the files Debug.py, Location.py, and Word.py are in the same folder. The program can be executed as follows (using animals.txt as input, for example):

```
python3 WordSearch.py < animals.txt
```

It may help to discuss the source code for this program. Let's do it in segments. First, the library imports:

```
# import libraries
from Location import Location
from Word import Word
from sys import stdin
from random import sample, choice, randint
```

The library imports include the *Location* and *Word* classes that you will design. They also include support for reading from *stdin*, and for performing the following random functions:

- *sample(lst, n)*: returns a list of *n* elements randomly chosen from *lst*.
- *choice(lst)*: returns a single item randomly chosen from *lst*.
- *randint(x, y)*: returns a randomly chosen integer within the range [x, y].

These three functions will be further described as they are encountered in the source code. The next segment defines two constants:

```
# define constants
NUM_WORDS = 15  # how many words to randomly select from the input
GRID_SIZE = 25  # the height/width of the "fictitious" grid
```

NUM_WORDS will determine how many words are randomly chosen from the ones provided via *stdin*. Note that this value should be less than or equal to the total number of words provided! *GRID_SIZE* provides a fictitious grid size (since there is no actual grid for this programming assignment), and restricts the word locations defined later.

Next, the main part of the program:

```
######
# MAIN
######
# read the words from stdin
words = [ ]
for line in stdin:
    # remove the trailing newline and convert to uppercase
    words.append(line.rstrip("\n").upper())
```

At the beginning of the main part of the program, a list of words is initialized. Next, the words inputted via *stdin* are stored into the list. Note that the trailing newline is stripped from each word, and the words are converted to uppercase. A random sampling of the words is then taken:

```
# grab a sampling of the specified number of words
words = sample(words, NUM_WORDS)
```

This statement randomly selects *NUM_WORDS* words from the list *words* and stores them back into the same list! This is fine since we will not be needing the rest of the words again. The list *words* is simply overwritten with the randomly chosen words.

Next, *Word* instances are created for each word:

```
# initialize a list of Word instances
word_objects = [ ]
# for each word, randomly pick an orientation and Location
for word in words:
    # randomly pick an orientation
    orientation = choice(Word.ORIENTATIONS)
```

```
        # randomly pick a Location (within the grid)
        row = randint(0, GRID_SIZE - 1)
        col = randint(0, GRID_SIZE - 1)
        location = Location(row, col)

        # append an Word instance of this word
        word_objects.append(Word(word, orientation, location))
```

First, a list of *`Word`* objects is initialized. For each word, a random orientation is chosen; specifically, a random choice from the class variable *`ORIENTATIONS`* is made. Since *`ORIENTATIONS`* is a class variable within the *`Word`* class, it must be accessed using its fully qualified name (i.e., *`Word.ORIENTATIONS`*). Next, a random *`Location`* is selected (by specifying random values for *`row`* and *`col`* to be within *`GRID_SIZE`*). More precisely, an integer in the range `[0, GRID_SIZE - 1]` is selected for *`row`* and *`col`*. If *`GRID_SIZE`* is 10, for example, then *`row`* can take on a random value between 0 and 9 inclusive. Finally, a *`Word`* instance is created with the current *`word`*, *`orientation`*, and *`Location`*, and is appended to the list of word objects.

Finally, the words are displayed:

```
    # display the words
    for word in word_objects:
        print(word)
```

Constraints and notes

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

- You must submit a ZIP file containing Location.py and Word.py. These filenames are required. The class names *`Location`* 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. Pay attention to the differences in output depending on the value of the variable *`DEBUG`* defined in Debug.py.
- 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 the contents of WordSearch.py to Word.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 provided files (i.e., Debug.py, WordSearch.py, animals.txt, and words.txt).
- In the `Location` class, the constructor should take `row` and `col` as parameters. Moreover, you must specify default parameter values for `row` and `col` (both 0). The constructor should set appropriate values for the instance variables to properly instantiate a `Location` object.
- In the `Location` class, you must implement getters and setters for the instance variables `_row` and `_col`.
- In the `Location` class, you must implement range checking in the setters for the instance variables `_row` and `_col` such that they are greater than or equal to 0. If a negative value is provided, it should default to 0.
- 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, the constructor should take `word`, `orientation`, and `location` as parameters. Moreover, you must specify default parameter values for `orientation` and `location` (both *None*). The constructor should set appropriate values for the instance variables to properly instantiate a `Word` object.
- In the `Word` class, you must implement getters and setters for the instance variables `_word`, `_orientation`, and `_location`.

Deliverable

Submit a ZIP file containing the **two** Python 3 source files (Location.py and Word.py) representing the two classes (`Location` and `Word`). They will be combined with the provided source files (Debug.py and WordSearch.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 10, *GRID_SIZE* is 25)

```
SHEEP
BADGER
KANGAROO
PARROT
ARMADILLO
FROG
PLATYPUS
CROCODILE
PANTHER
FLAMINGO
```

Sample 2 (*DEBUG* is True, *NUM_WORDS* is 10, *GRID_SIZE* is 25)

```
CHEETAH/VU@(6,16)
PANTHER/HL@(7,7)
IGUANA/DRD@(17,13)
HYENA/HR@(15,14)
KANGAROO/DLD@(13,12)
FROG/DLD@(10,11)
ARMADILLO/DLD@(24,5)
EAGLE/HR@(21,2)
TURTLE/VU@(11,3)
CHAMAELEON/VU@(15,4)
```

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 *_row* and *_col* in the *Location* class are greater than or equal to 0) in the setters, use them (the setters) when initializing these values 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.

- Make sure to declare *ORIENTATIONS* as a class variable in the *Word* class.
- Range checking is not required for any instance variable in the *Word* class.

Rubric

Please note the following rubric for this programming assignment:

Word Search (part 1)		
#	**Item**	**Points**
1	ZIP file submitted and contains Location.py and Word.py	2
2	Good coding style	2
3	Appropriate comments	2
4	Appropriate header	2
5	Meaningful variable names	2
6	Input is from stdin	3
7	Output is correct and properly responds to DEBUG	2
8	Location constructor properly instantiates a Location	3
9	Location constructor correctly uses default parameters	3
10	Location getters are properly included	4
11	Location setters are properly included with range checking	4
12	Orientations are properly defined through a class variable in Word	3
13	Word constructor properly instantiates a Word	5
14	Word constructor correctly uses default parameters	3
15	Word getters are properly included	4
16	Word setters are properly included	4
17	Proper naming convention for instance variables	2
TOTAL		**50**