

```

1  ''' Implementation of a non-static version of the game Word Search Puzzle.
2
3  ...
4  from random import shuffle, randint, sample
5  from string import ascii_uppercase
6  import xml.etree.ElementTree as ET
7  import glob, os, time, re
8
9  __author__ = "Oscar Guillen, Patricia Reinoso"
10 __date__ = "17/02/2017"
11 __version__ = "1"
12
13 commands = ["help", "exit", "rotate", "find"]
14 bool_answers = ["yes", "no"]
15 directions = ["right", "left", "up", "down"]
16 messages = {
17     "already_found_word": " was already found.",
18     "choose_subject": "Please, choose a subject...\n",
19     "error": "\nAn error occurred. Leaving the game.\n",
20     "error_instruction": "\nAn error occurred. No instructions. "\
21         "Leaving the game.\n",
22     "enter_to_continue": "\nEnter to Continue.",
23     "exit_game": "\nLeaving the game.\n",
24     "final_cell": "\nPlease, insert the coordinates of the final cell:",
25     "good_word": "\nCongratulations! You just found ",
26     "highscore": "NEW HIGHSCORE",
27     "incorret_format": "\nERROR: Incorrect format. It was not added.",
28     "incorret_format_clue": "\nERROR: Incorrect format. It was not "\
29         "selected.",
30     "incorret_word": " is not on the list.",
31     "initial_cell": "\nPlease, insert the coordinates of the initial"\
32         " cell:",
33     "insert_column_number": "Column number ? ",
34     "insert_command": "\nAction [rotate|find|help|exit] ? ",
35     "insert_direction": "Direction [up|down|right|left] ? ",
36     "insert_menu_action": "\nAction [p|h|e] ? ",
37     "insert_number_spaces": "Spaces ? ",
38     "insert_row_number": "Row number ? ",
39     "invalid_cell": "\nERROR: That cell is not inside the board.",
40     "invalid_column": "\nERROR: That column is not inside the board.",
41     "invalid_command": "\nERROR: That is an invalid command. Only "\
42         "'rotate', 'find', 'help', 'exit' are possible.\n",
43     "invalid_direction": "\nERROR: That is an invalid direction. Only "\
44         "'right', 'left', 'up', 'down' are possible.",
45     "invalid_menu_option": "\nERROR: Invalid option. Only 'p' for play, "\
46         "'h' for help and 'e' for exit.",
47     "invalid_number_spaces": "\nERROR: Invalid number of spaces.",
48     "invalid_play_again_option": "\nERROR: Invalid option. Only 'yes'\
49         " or 'no'.",
50     "invalid_row": "\nERROR: That row is not inside the board.",
51     "invalid_subject": "\nERROR: It is not a valid subject. Try "\
52         "again.\n",
53     "it_is_not_a_line": "\nERROR: The cells must form a vertical or a "\
54         "horizontal line of at least 3 letters.",
55     "it_is_not_an_integer": "\nERROR: It must be an integer.",
56     "menu": "MENU\n\n[p] Play\n[h] Help\n[e] Exit\n",
57     "play_again": "\n\nPlay again [yes|no] ? ",
58     "separator": "\n-----"\
59         "-----\n",
60     "setting_up": "Setting up...",
61     "single_player_modes": "MODE\n\nPractice mode\n",
62     "starting": "Starting...",
63     "get_subject": "Type here your subject: ",
64     "welcome": "WORD SEARCH PUZZLE\n\nLet's play!\n",
65     "win": "\n\nCongratulations! You found all the words. You won. ",
66     "winner": " is the winner.",
67     "words_to_find": "Words to find: "
68 }
69
70 class Instruction:
71     ''' Manage all the information corresponding to the rules of
72         the game.
73     ...
74
75     def __init__(self):
76         self.instruction = None
77
78     def import_instruction(self, file):
79         ''' Import the set of instructions of the game from a file.
80
81         Parameters
82         -----
83         file : str
84             Path of the file where the instructions are found.
85
86         ...
87
88         try:
89             f = open(file, 'r')
90             self.instruction = f.read()
91
92         except:
93             print(messages["error_instruction"])
94             exit()
95
96     def display(self):
97         '''Display on the standard output the rules of the game.
98
99         ...
100         print(self.instruction)

```

```

84 class Board:
85     ''' Manage the board where the words to be found are placed.
86
87     '''
88     def __init__(self):
89         self.original_grid = [[]]
90         self.current_grid = [[]]
91         self.rows = 10
92         self.columns = 10
93         self.min_word_length = 3
94         self.max_word_length = 8
95         self.max_num_words = 10
96     def build (self, words):
97         ''' Add to the board all the words given and fill the blanks spaces left
98             with random letters .
99
100         Parameters
101         -----
102             words : [str]
103                 An array with the words to be placed on the board.
104
105         '''
106         self.current_grid = [['' for x in range(10)] for y in range(10)]
107         num_words = len(words)
108         words.sort(key=len)
109         words.reverse()
110         cols = [0,1,2,3,4,5,6,7,8,9]
111         rows = [0,1,2,3,4,5,6,7,8,9]
112         for word in words:
113             added = False
114             while not added:
115                 position = randint(0,1) # 0 is row, 1 is column.
116                 if position and len(cols) > 0:
117                     num = sample(cols,1)[0]
118                     added = self.add_word_into_row(num,word)
119                     if added:
120                         cols.remove(num)
121                 elif not position and len(rows) > 0:
122                     num = sample(rows,1)[0]
123                     added = self.add_word_into_column(num,word)
124                     if added:
125                         rows.remove(num)
126             self.fill()
127
128         # Copy the built list.
129         self.original_grid = []
130         for row in self.current_grid:
131             self.original_grid.append(list(row))
132
133     def is_valid_row(self,row):
134         """ Check if the row number is inside the grid.
135
136         Parameters
137         -----
138             row : int
139                 row number.
140
141         Returns
142         -----
143             bool
144                 True for success, False otherwise.
145
146         """
147         return (row < self.rows and row >= 0)
148
149     def is_valid_column(self,column):
150         """ Check if the column number is inside the grid.
151
152         Parameters
153         -----
154             column : int
155                 column number.
156
157         Returns
158         -----
159             bool
160                 True for success, False otherwise.
161
162         """
163         return (column < self.columns and column >= 0)
164
165     def is_valid_number_spaces(self,direction,number):
166         """ Check if the number is to rotate is correct.
167
168         Parameters
169         -----
170             direction : str
171                 direction to rotate.
172             number : int
173                 number of spaces to rotate.
174
175         Returns
176         -----
177             bool
178                 True for success, False otherwise.
179
180         """
181         if number > 0 and \
182             ((direction == "up" or direction == "down") and \
183              number < self.columns) or \
184             ((direction == "left" or direction == "right") and \
185              number < self.rows)):

```

```

167         return False        return True
168
169     def rotate_horizontally(self, row, number, direction):
170         """ Rotates a row of the current grid a 'number' amount of spaces.
171
172         Parameters
173         -----
174         row : int
175             row number to rotate.
176         number : int
177             number of space to rotate.
178         direction : str
179             direction to rotate the row ("left" or "right").
180
181         Returns
182         -----
183         bool
184             True if the board was successfully rotate, False otherwise.
185
186         """
187         if self.is_valid_row(row):
188             number = number % self.rows
189             if direction == "right":
190                 self.current_grid[row] = self.current_grid[row][-number:] +\
191 self.current_grid[row][:number]
192             elif direction == "left":
193                 self.current_grid[row] = self.current_grid[row][number:] +\
194 self.current_grid[row][:number]
195             return True
196         return False
197
198     def rotate_vertically(self, column, number, direction):
199         """ Rotates a column of the current grid a 'number' amount of spaces.
200
201         Parameters
202         -----
203         column : int
204             column number to rotate.
205         number : int
206             number of space to rotate.
207         direction : str
208             direction to rotate the column ("up" or "down").
209
210         Returns
211         -----
212         bool
213             True if the board was successfully rotate, False otherwise.
214
215         """
216         if self.is_valid_column(column):
217             number = number % self.columns
218             temp_col=[]
219             for i in range(self.columns):
220                 temp_col.append(self.current_grid[i][column])
221
222             if direction == "up":
223                 temp_col = temp_col[number:] + temp_col[:number]
224             elif direction == "down":
225                 temp_col = temp_col[-number:] + temp_col[:-number]
226
227             for i in range(self.columns):
228                 self.current_grid[i][column]=temp_col[i]
229             return True
230         return False
231
232     def get_letter(self, row, column):
233         """ Return the letter that inside a cell on the board.
234
235         Parameters
236         -----
237         row : int
238             row of the cell of the letter wanted.
239         column : int
240             column of the cell of the letter wanted.
241
242         Returns
243         -----
244         str
245             Letter extratect from the cell of the board.
246
247         """
248         if self.is_valid_cell(row, column):
249             return self.current_grid[row][column]
250         return None
251
252     def display(self):
253         """ Display the grid on the standard output. Row and columns numbers
254         are included.
255
256         """
257         print("\n  ", end='')
258         for i in range (self.columns):
259             print (str(i) + "  ", end='')
260         print("\n")
261         for i in range (self.rows):
262             for j in range (self.columns):
263                 if j == 0:
264                     print (str(i) + "  ", end='')
265                 print (self.get_letter(i,j) + "  ", end='')
266             if j == self.rows - 1:
267                 print (str(i), end='')

```

```

250         print("\n")
251     print (" ", end='')
252     for i in range (self.columns):
253         print (str(i) + " ", end='')
254     print("\n")
255
256     def is_valid_cell(self,row,column):
257         """ Check if the cell is inside the grid.
258
259         Parameters
260         -----
261         row : int
262             row number of the cell.
263         column : int
264             column number of the cell.
265
266         Returns
267         -----
268         bool
269             True for success, False otherwise.
270
271         """
272         if self.is_valid_row(row) and self.is_valid_column(column):
273             return True
274         return False
275
276     def can_generate_a_word(self,row1,column1,row2,column2):
277         """ Receive 2 cells and check they are valid, if they form a vertical
278             or an horizontal line, and if they form a word long enough.
279
280         Parameters
281         -----
282         row1 : int
283             row number of the starting cell.
284         column1 : int
285             column number of the starting cell.
286         row2 : int
287             row number of the final cell.
288         column2 : int
289             column number of the final cell.
290
291         Returns
292         -----
293         bool
294             True for success, False otherwise.
295
296         """
297         if (self.is_valid_cell(row1,column1) and
298             self.is_valid_cell(row2,column2) and \
299
300             (row1 == row2 and column1 != column2 and
301              abs(column1 - column2) + 1 >= self.min_word_length) or \
302
303             (column1 == column2 and row1 != row2 and \
304              abs(row1 - row2) + 1 >= self.min_word_length)):
305             return True
306         return False
307
308     def get_word(self,row1,column1,row2,column2):
309         """ Get the word from the grid. The word is specified by a starting cell
310             and a final cell.
311
312         Parameters
313         -----
314         row1 : int
315             row number of the starting cell.
316         column1 : int
317             column number of the starting cell.
318         row2 : int
319             row number of the final cell.
320         column2 : int
321             column number of the final cell.
322
323         Returns
324         -----
325         str
326             Word obtained from the grid. None if the cells are not valid.
327
328         """
329         if self.can_generate_a_word(row1,column1,row2,column2):
330             word = ""
331             # The word is placed horizontally
332             if (row1 == row2):
333                 start = min(column1,column2)
334                 end = max(column1,column2)
335                 for i in range (start,end+1):
336                     word = word + self.get_letter(row1,i)
337                 return word
338             # The word is placed vertically
339             elif (column1 == column2):
340                 start = min(row1,row2)
341                 end = max(row1,row2)
342                 for i in range (start,end+1):
343                     word = word + self.get_letter(i,column1)
344                 return word
345             else:
346                 return None

```

```

333 def add_word_into_row(self, row, word):
334     ''' Insert a word into a row of the board.
335
336     Parameters
337     -----
338     row: int
339         The number of the row.
340     word: str
341         The word to be inserted.
342
343     Returns
344     -----
345     bool
346         True for success, False otherwise.
347
348     '''
349     column = randint(0,9)
350     aux_column = column
351     # Checking the possibility to add the word
352     for letter in word:
353         if aux_column > 9:
354             aux_column = 0
355         if self.current_grid[row][aux_column] != '':
356             if self.current_grid[row][aux_column] != letter:
357                 return False
358         aux_column += 1
359     # Adding the word.
360     for letter in word:
361         if column > 9:
362             column = 0
363         if self.current_grid[row][column] == '':
364             self.current_grid[row][column] = letter
365         column += 1
366     return True
367
368 def add_word_into_column(self, column, word):
369     ''' Insert a word into a column of the board.
370
371     Parameters
372     -----
373     column: integer
374         The number of the column.
375     word: string
376         The word to be inserted.
377
378     Returns
379     -----
380     bool
381         True for success, False otherwise.
382
383     '''
384     row = randint(0,9)
385     aux_row = row
386     # Checking the possibility to add the word
387     for letter in word:
388         if aux_row > 9:
389             aux_row = 0
390         if self.current_grid[aux_row][column] != '':
391             if self.current_grid[aux_row][column] != letter:
392                 return False
393         aux_row += 1
394     # Adding the word.
395     for letter in word:
396         if row > 9:
397             row = 0
398         if self.current_grid[row][column] == '':
399             self.current_grid[row][column] = letter
400         row += 1
401     return True
402
403 def fill(self):
404     ''' Fill the blank spaces of the board with random letters.
405
406     Parameters
407     -----
408
409     Returns
410     -----
411     bool
412         True for success, False otherwise.
413
414     '''
415     alfabet = ascii_uppercase
416     for i in range(10):
417         for j in range(10):
418             if self.current_grid[i][j] == '':
419                 self.current_grid[i][j] = sample(alfabet,1)[0]
420
421 def get_max_num_words(self):
422     ''' Return the maximun number of words allowed.
423
424     Returns
425     -----
426     int
427         The maximun number of words.
428
429     '''
430     return self.max_num_words
431
432 def get_min_length_word(self):
433     ''' Return the Minimun length of a word allowed.
434
435     Returns
436     -----
437     int
438         The minimun length.
439
440     '''
441     return self.min_word_length
442
443 def get_max_length_word(self):
444     ''' Return the Maximun length of a word allowed.

```

```

416         Returns
417         -----
418         int
419             The maximun length
420
421         '''
422         return self.max_word_length
423
424 class Subject:
425     ''' Manage the subjects all the words on a board are related to. '''
426
427     def __init__(self):
428         self.name = None
429         self.content = []
430
431     def add_word(self, word):
432         ''' Add a word into the subject.
433
434         Parameters
435         -----
436         word : str
437             Word to be added.
438
439         '''
440
441         if word is not None:
442             self.content.append(word)
443
444     def import_subject(self, file):
445         ''' Import the content of a well-formed xml into a subject.
446
447         Parameters
448         -----
449         file : str
450             The path of the file.
451
452         '''
453
454         # Parsing XML
455         # xml.etree.ElementTree from Python The ElementTree XML API
456         # Source code: https://docs.python.org/3.4/library/xml.etree.elementtree.html
457         tree = ET.parse(file)
458         root = tree.getroot()
459         words = 0
460         if len(root) > 0:
461             self.name = root[0].attrib['name']
462             root = root[0]
463             for child in root:
464                 if child.tag == "word":
465                     new_word = child.text
466                     # Parsing a string using Regular Expression
467                     # re from Python Regular expression operations
468                     # Source code: https://docs.python.org/3.4/library/re.html
469                     if re.match('[a-zA-Z]+$', new_word) is not None:
470                         self.add_word(new_word)
471                         words += 1
472
473         else:
474             return False
475         return words >= 12
476
477     def get_name(self):
478         ''' Return the name of the subject.
479
480         Returns
481         -----
482         str
483             Name of the subject.
484
485         '''
486         return self.name
487
488     def get_words(self):
489         ''' Return all the words of the subject.
490
491         Returns
492         -----
493         [str]
494             All the words of the subject. Can be empty.
495
496         '''
497         return self.content
498
499 class Dictionary:
500     ''' Manage all the subjects available on the game.
501
502     '''
503
504     def __init__(self):
505         self.subjects = []
506
507     def add_subject(self, subject):
508         ''' Add a subject object into the dictionary.
509
510         Parameters
511         -----
512         subject: Subject
513             Subject to be added.
514
515         '''
516         self.subjects.append(subject)

```

```

499
500 def display_subjects(self):
501     ''' Display on the standard output the list of the names of the
502     all the subjects.
503     '''
504     print('Subjects: ')
505     for subject in self.subjects:
506         print('- ' + subject.get_name())
507     print('')
508
509 def load(self):
510     ''' Load the dictionary from xml files into subjects directory.
511     '''
512     Returns
513     -----
514     bool
515     True for success, False otherwise.
516     '''
517     # Manage files and directories.
518     # os.walk from Python Miscellaneous operating system interfaces.
519     # Source code: https://docs.python.org/3.4/library/os.html#os.walk
520     try:
521         num_of_correct_subjects = 0
522         for root, dirs, files in os.walk("subjects"):
523             for file in files:
524                 if file.endswith(".xml"):
525                     new_subject = Subject()
526                     correct = new_subject.import_subject(os.path.join(root, \
527                                                                 file))
528                     if correct:
529                         self.add_subject(new_subject)
530                         num_of_correct_subjects += 1
531         return num_of_correct_subjects > 0
532     except:
533         return False
534
535 def get_subject_by_name(self, name):
536     ''' Return a subject using its name element.
537     '''
538     Parameters
539     -----
540     name : str
541     The name of the subject as a string.
542     '''
543     Returns
544     -----
545     Subject
546     Subject corresponding to 'name'.
547     '''
548     for subject in self.subjects:
549         if subject.get_name().lower() == name.lower():
550             return subject
551     return None
552
553 class Clue:
554     ''' Manage the list of words that the player needs to find on the game.
555     '''
556     '''
557     def __init__(self):
558         self.subject_name = None
559         self.words_not_found = []
560         self.words_found = []
561
562     def build(self, subject, num_words, min_length, max_length):
563         ''' Build the clue, using the words of a subject, adding randomly
564         the words into the not found list. There will be 10 clues per board
565         by default.
566         '''
567         Parameters
568         -----
569         subject : Subject
570         Subject object that contain the words to be used.
571         num_words : int
572         Number of words to be selected.
573         min_length : int
574         Minimun length of a word.
575         max_length: in
576         Maximun length of a word.
577         '''
578         self.subject_name = subject.get_name()
579         words = list(subject.get_words())
580         # Functions from Python Random Lib.
581         shuffle(words)
582         if len(words) < num_words:
583             lim = len(words)
584         elif num_words <= 0:
585             lim = 10
586         else:
587             lim = num_words
588
589         i = 0
590         while lim > 0 and i < len(words):
591             if len(words[i]) <= max_length and len(words[i]) >= min_length:
592                 self.add_word_to_not_found(words[i].upper())
593                 lim -= 1
594             i += 1

```

```

582         # Inconsistency
583         if lim > 0:
584             print(messages["error"])
585             exit()
586
587     def word_already_found(self, word):
588         """ Checks if a word is on the list of words already found.
589
590         Parameters
591         -----
592         word : str
593             Word to check.
594
595         Returns
596         -----
597         bool
598             True for success, False otherwise.
599
600         """
601         return (word in self.words_found)
602
603     def word_not_found(self, word):
604         """ Cheks if a word is on the list of words that have not been found.
605
606         Parameters
607         -----
608         word : str
609             word to be checked.
610
611         Returns
612         -----
613         bool
614             True for success, False otherwise.
615
616         """
617         return (word in self.words_not_found)
618
619     def word_in_clue(self, word):
620         """ Cheks if a word belongs to a clue. Checks if a word is included on
621         the list of words already found or on the list of words that have
622         not been found.
623
624         Parameters
625         -----
626         word : str
627             word to be checked.
628
629         Returns
630         -----
631         bool
632             True for success, False otherwise.
633
634         """
635         return (self.word_already_found(word) or self.word_not_found(word))
636
637     def add_word_to_not_found(self, word):
638         """ Add a word to the clue as not found.
639
640         Parameters
641         -----
642         word : string
643             Word that will be added.
644
645         """
646         if word == "" or word == None:
647             print(messages["incorret_format"])
648         else:
649             self.words_not_found.append(word)
650
651     def add_word_to_found(self, word):
652         """ Add a word to the list of words that have been found.
653
654         Parameters
655         -----
656         word (str): word to be added.
657
658         """
659         self.words_found.append(word)
660
661     def remove_word_from_not_found(self, word):
662         """ Check if the word is on the list of words that have not been found
663         and removed it.
664
665         Parameters
666         -----
667         word : str
668             Word to be removed.
669
670         Returns
671         -----
672         bool
673             True if the word was successfully removed. False otherwise.
674
675         """
676         if (self.word_not_found(word)):
677             self.words_not_found.remove(word)
678             return True
679         return False
680
681     def found_all_the_words(self):
682         """ Check if the list of words that have not been found is empty and
683         the list of words that have been found is not empty.
684
685         Returns
686         """

```



```

665         -----
666         bool
667         True for success, False otherwise.
668
669         """
670         return ((len(self.words_not_found) == 0) and\
671                 (len(self.words_found) != 0))
672
673     def get_words_not_found(self):
674         """ Return the list of words not found.
675
676         Returns
677         -----
678         [str]
679         List of words that has not been found by the player.
680
681         """
682         return self.words_not_found
683
684 class BySolution(Clue):
685     """ Subclass that displays all the words in the Clue.
686
687     """
688     def display(self):
689         """ Display on the standard output the name of the subject and the
690         complete list of words that the player needs to find.
691
692         """
693         print(self.subject_name + ": ")
694         print(self.words_not_found)
695
696 class Game:
697     """ Superclass. Manage information of a game on its different modes.
698
699     """
700     def __init__(self):
701         self.board = None
702         self.clue = None
703         self.start_again = False
704
705     def add_board(self, board):
706         """ Associate a built board to the game.
707
708         Parameters
709         -----
710         board : Board
711             Board to be added to the game.
712
713         """
714         self.board = board
715
716     def select_subject(self):
717         """ Return a selected subject from the dictionary.
718
719         Returns
720         -----
721         Subject
722             Subject selected by the player.
723
724         """
725         sname = ""
726         subject = None
727         while subject == None:
728             print(messages["choose_subject"])
729             dictionary.display_subjects()
730             sname = input(messages["get_subject"])
731             subject = dictionary.get_subject_by_name(sname)
732             if subject == None:
733                 print(messages["separator"])
734                 print(messages["invalid_subject"])
735
736         return subject
737
738     def select_clue(self, subject, num_words, min_length, max_length):
739         """ Select a specific type of clue. For this version, it will be
740         BySolution.
741
742         Parameters
743         -----
744         subject : Subject
745             subject that will be used into the building clue process.
746         num_words : int
747             Number of words to be selected
748         min_length : int
749             Minimum length of a word.
750         max_length: int
751             Maximum length of a word.
752
753         """
754         clue = BySolution()
755         if subject == None:
756             print(messages["incoret_format_clue"])
757         else:
758             clue.build(subject, num_words,min_length, max_length)
759             self.clue = clue
760
761     def get_row_number(self):
762         """ Read from standard input an integer.
763
764         Returns
765         -----

```

```

748         int : row number read.
749
750     """
751     is_valid = False
752     while not is_valid:
753         try:
754             n = int(input(messages["insert_row_number"]))
755             if self.board.is_valid_row(n):
756                 return n
757             print(messages["invalid_row"])
758         except:
759             print(messages["it_is_not_an_integer"])
760
761     def get_column_number(self):
762         """ Read from standard input an integer.
763
764         Returns
765         -----
766         int : column number read.
767
768     """
769     is_valid = False
770     while not is_valid:
771         try:
772             n = int(input(messages["insert_column_number"]))
773             if self.board.is_valid_column(n):
774                 return n
775             print(messages["invalid_column"])
776         except:
777             print(messages["it_is_not_an_integer"])
778
779     def find_word(self):
780         """ Find a word specified by the user. Get the row and the column
781         numbers corresponding to initial and the final cells of the word.
782         Get the word from the board, and check if the word is valid.
783         Display appropriate messages to the user.
784
785     """
786     print(messages["initial_cell"])
787     row1 = self.get_row_number()
788     column1 = self.get_column_number()
789     print(messages["final_cell"])
790     row2 = self.get_row_number()
791     column2 = self.get_column_number()
792     word = self.board.get_word(row1, column1, row2, column2)
793     if word != None:
794         if not self.clue.word_in_clue(word):
795             print("\n" + word + " " + messages["incoret_word"])
796             return
797         if self.clue.word_already_found(word):
798             print(" " + word + " " + messages["already_found_word"])
799         else:
800             self.clue.add_word_to_found(word)
801             self.clue.remove_word_from_not_found(word)
802             print(messages["good_word"] + " " + word + ".")
803     else:
804         print(messages["it_is_not_a_line"])
805
806     def setup(self):
807         """ Setup all the configuration of the game before starting. """
808
809         new_board = Board()
810         selected_subject = self.select_subject()
811         self.select_clue(selected_subject,
812                             new_board.get_max_num_words(),
813                             new_board.get_min_length_word(),
814                             new_board.get_max_length_word())
815
816         words = list(self.clue.get_words_not_found())
817         new_board.build(words)
818         self.add_board(new_board)
819
820     def get_direction(self):
821         """ Read from standard input a string corresponding to a direction.
822         Only 'right', 'left', 'up', 'down' are possible.
823
824         Returns
825         -----
826         str
827             direction read.
828
829     """
830     is_valid = False
831     while not is_valid:
832         direction = (input(messages["insert_direction"])).lower()
833         if direction in directions:
834             return direction
835         print(messages["invalid_direction"])
836
837     def get_number_spaces(self, direction):
838         """ Read from standard input an integer.
839
840         Parameters
841         -----
842         direction : str
843             Direction the board is going to rotate to.
844
845         Returns
846         -----
847         int
848             Number of spaces read.
849
850     """
851     is_valid = False

```

```

831         while not is_valid:
832             try:
833                 n = int(input(messages["insert_number_spaces"]))
834                 if self.board.is_valid_number_spaces(direction,n):
835                     return n
836             except:
837                 print(messages["it_is_not_an_integer"])
838
839     def get_rotation_attr(self):
840         """ Get the direction to rotate the board, the number of row or column
841         that is going to rotate and the amount of spaces to rotate. Then
842         rotates the board accordint to these parameters.
843
844         """
845         direction = self.get_direction()
846         if direction == "up" or direction == "down":
847             column = self.get_column_number()
848             number = self.get_number_spaces(direction)
849             self.board.rotate_vertically(column,number,direction)
850         elif direction == "left" or direction == "right":
851             row = self.get_row_number()
852             number = self.get_number_spaces(direction)
853             self.board.rotate_horizontally(row,number,direction)
854
855     def get_play_again(self):
856         """ Ask the user if he-she wants to start a new game.
857
858         Returns
859         -----
860         bool
861             True if success, False otherwise.
862
863         """
864         while True:
865             answer = (input(messages["play_again"])).lower()
866             if answer in bool_answers:
867                 if answer == "yes":
868                     return True
869                 elif answer == "no":
870                     return False
871             print(messages["invalid_play_again_option"])
872
873     def end_current_game(self):
874         """ If the player wants to end the game, sets the variable start_again
875         to True. Otherwise exit the game.
876
877         """
878         if self.get_play_again():
879             self.start_again = True
880         else:
881             exit_game()
882
883     def read_command(self):
884         """ Reads a command from standard input. Only 'rotate', 'word', 'help',
885         'exit' are valid. Calls the corresponding function to execute each
886         command.
887
888         """
889         is_valid = False
890         while not is_valid:
891             command = (input(messages["insert_command"])).lower()
892             if command in commands:
893                 if command == "help":
894                     inst.display()
895                 elif command == "exit":
896                     self.end_current_game()
897                 elif command == "rotate":
898                     self.get_rotation_attr()
899                 elif command == "find":
900                     self.find_word()
901                 return None
902             else:
903                 print(messages["invalid_command"])
904
905     def turn(self):
906         """ Manages the turns of the player. Call the function to read the
907         commands, and display the current state of the game.
908
909         """
910         print (messages["separator"])
911         self.board.display()
912         self.clue.display()
913         self.read_command()
914
915     def play_again(self):
916         """ Return if the player wants to play again.
917
918         Returns
919         -----
920         bool
921             True if the player wants to play again. False otherwise.
922
923         """
924         return self.start_again
925
926 class PracticeMode(Game):
927     """ Single player mode of game implemented on version 1.
928     Subclass managing specific behaviours of the game.
929
930     """

```

```

914     def is_win(self):
915         ''' Check the winning condition.
916
917         Returns
918         -----
919         bool
920
921             True if success, False otherwise.
922
923         '''
924         return (self.clue.found_all_the_words())
925
926     def play(self):
927         '''
928         Manage the gameplay. Handle the turns.
929
930         '''
931         self.start_again = False
932         while not self.start_again:
933             self.turn()
934             if self.is_win():
935                 print(messages["win"])
936                 self.end_current_game()
937
938 # Dictionary of words.
939 dictionary = Dictionary()
940
941 # Class game.
942 game = PracticeMode()
943
944 # Instructions of the game
945 inst = Instruction()
946
947 def configure_instructions():
948     ''' Import the instructions into the program.
949
950     '''
951
952     inst.import_instruction("./rules/instructions.txt")
953
954 def display_initial_message():
955     ''' Display on the standard output a welcome message.
956
957     '''
958     print (messages["welcome"])
959
960 def display_menu():
961     ''' Display on the standard output a welcome message.
962
963     '''
964     print (messages["menu"])
965
966 def start_game():
967     ''' Start the game. '''
968
969     os.system('clear')
970     display_initial_message()
971     input(messages["enter_to_continue"])
972     os.system('clear')
973
974     option = ''
975     while(option != 'p'):
976         display_menu()
977         option = input(messages["insert_menu_action"])
978         option = option.lower()
979         os.system('clear')
980         if option == 'h':
981             inst.display()
982             print(messages["separator"])
983         elif option == 'e':
984             messages["exit_game"]
985             exit()
986         elif option != 'p':
987             print(messages["invalid_menu_option"])
988
989     print(messages["setting_up"])
990     time.sleep(2)
991     os.system('clear')
992     game.setup()
993     os.system('clear')
994     print(messages["starting"])
995     time.sleep(2)
996     os.system('clear')
997     game.play()
998
999 def exit_game():
1000     ''' Display an exit message and exit the program.
1001
1002     '''
1003     print (messages["exit_game"])
1004     exit()
1005
1006 def main():
1007     ''' Main function of the program
1008
1009     '''
1010
1011     configure_instructions()
1012     correct = dictionary.load()
1013     if not correct:
1014         print(messages["error"])
1015         exit()
1016     play_again = True
1017     while play_again:
1018         start_game()
1019         play_again = game.play_again()

```

```
997
998 if __name__ == '__main__':
999     main()
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
```