Assignment 2

COMP9021, Trimester 3, 2024

1 General matters

1.1 Aim

The purpose of the assignment is to:

- develop your problem solving skills;
- design and implement the solutions to problems in the form of medium sized Python programs;
- practice the design and implementation of search techniques, with recursion as a good approach;
- design and implement an interface based on the desired behaviour of an application program;
- possibly practice using the `re` and `numpy` modules.

1.2 Submission

Your program will be stored in a file named `crossword.py`. After you have developed and tested your program, upload it using Ed (unless you worked directly in Ed). Assignments can be submitted more than once; the last version is marked. Your assignment is due by November 18, 10:00am.

1.3 Assessment

The assignment is worth 13 marks. It is going to be tested against a number of inputs. For each test, the automarking script will let your program run for 30 seconds.

Assignments can be submitted up to 5 days after the deadline. The maximum mark obtainable reduces by 5% per full late day, for up to 5 days. Thus if students A and B hand in assignments worth 12 and 11, both two days late (that is, more than 24 hours late and no more than 48 hours late), then the maximum mark obtainable is 11.7, so A gets $\min(11.7, 12) = 11.7$ and B gets $\min(11.7, 11) = 11$.

The outputs of your programs should be **exactly** as indicated.

1.4 Reminder on plagiarism policy

You are permitted, indeed encouraged, to discuss ways to solve the assignment with other people. Such discussions must be in terms of algorithms, not code. But you must implement the solution on your own. Submissions are routinely scanned for similarities that occur when students copy and modify other people's work, or work very closely together on a single implementation. Severe penalties apply.

2 Solving crosswords

2.1 General description

The aim is to fill in a crossword grid, in which some letters might have been placed already, in one of two ways:

- being given the collection of all missing entries in an underlying solution to the crossword, find a way to place those entries in the grid (in one of possibly more than one way);
- being given a collection of words, find a way to select and place some of those words in the grid so that it solves the puzzle (in one of possibly more than one way).

We will consider grids with at least 2 rows and at least 2 columns, and such that any cell is next to another cell, though maybe not next to both horizontal and vertical cells; so any cell is part of a word of at least 2 letters, and by *word* we mean a word with at least 2 letters.

A presentation of a grid provided as input will be captured in a `.tex` file, and a presentation of a solved puzzle will also be captured in a `.tex` file; those `.tex` files can be given as arguments to `pdflatex` to generate `.pdf` files that display grids in a pleasing graphical form, but play no role in the assignment.

The code will be all about the design and implementation of a `Crossword` class. An object of class `Crossword` will provide an interface to work with a particular grid and collections of words of the kind just described; it will have methods for both problems previously described, and also benefit from the implementation of the `__str__()` special method to display some information about the grid and the words it possibly contains already.

`partial_grid_3.tex` is an example of a `.tex` file for a grid in which letters have been placed already, that can be considered as an input file, and here is the `.pdf` file created by `pdflatex` from this `.tex` file.

`solved_partial_grid_3.tex` is an example of a `.tex` file for a solution to that grid, that can be considered as an output file, and here is the `.pdf` file created by `pdflatex` from this `.tex` file.

The contents of all input `.tex` files will be of the following form:

```
\documentclass{standalone}
\usepackage{pas-crosswords}
\usepackage{tikz}

\begin{document}
\begin{tikzpicture}
...
\end{tikzpicture}
\end{document}
```

For input files, what lies between `\begin{tikzpicture}` and `\end{tikzpicture}` is of the following form:

```
\begin{crossgrid}[h=_, v=_]
\blackcases{_/_, ..., _/_}
\words[v]{_/_/_, ..., _/_/_}
\words[h]{_/_/_, ..., _/_/_}
\end{crossgrid}
```

Each of

- `\blackcases{_/_, ..., _/_}`,
- `\words[v]{_/_/_, ..., _/_/_}` and
- `\words[h]{_/_/_, ..., _/_/_}`

is optional and can appear in any order.

- In `\begin{crossgrid}[h=_, v=_]`, the first and second occurrences of `_` denote integers at least equal to 2, that represent the horizontal dimension and the vertical dimension of the grid, respectively; let *h-dim* and *v-dim* denote those dimensions, respectively.

- In `\blackcases{_/_, ..., _/_}`, each occurrence of `_/_` denotes a slash-separated pair of integers (i, j) with $1 \leq i \leq$ *h-dim* and $1 \leq j \leq$ *v-dim*, to denote that there is a black square at the intersection of the i^{th} column and the j^{th} row. If `\blackcases{_/_, ..., _/_}` is present then there is at least one such pair, consecutive pairs being separated by a comma.

- In `\words[v]{_/_/_, ..., _/_/_}`, each occurrence of `_/_/_` denotes a slash-separated triple (i, j, w) where i and j are integers with $1 \leq i \leq$ *h-dim* and $1 \leq j \leq$ *v-dim*, and where w is a word of at least 2 letters that is vertically placed in the grid and whose first letter is at the intersection of the i^{th} column and the j^{th} row. If `\words[v]{_/_/_, ..., _/_/_}` is present then there is at least one such triple, consecutive triples being separated by a comma.

- In `\words[h]{_/_/_, ..., _/_/_}`, each occurrence of `_/_/_` denotes a slash-separated triple (i, j, w) where i and j are integers with $1 \leq i \leq$ *h-dim* and $1 \leq j \leq$ *v-dim*, and where w is a word of at least 2 letters that is horizontally placed in the grid and whose first letter is at the intersection of the i^{th} column and the j^{th} row. If `\words[h]{_/_/_, ..., _/_/_}` is present then there is at least one such triple, consecutive triples being separated by a comma.

You can assume that the input is valid: no word or black square will start or extend beyond the dimensions of the grid.

For output files, what lies between `\begin{tikzpicture}` and `\end{tikzpicture}` is of the form `\gridcross{_, ... _}` where `_, ... _` is a comma-separated sequence of *v-dim* strings, each of length *h-dim*, to denote from top to bottom each of the rows of the filled grid, using `*`s to represent black squares, and using uppercase letters for the grid's entries.

In `.tex` files, there can be spaces between tokens almost everywhere (the `pas-crosswords` package actually prevents spaces to be used around some of the `/` characters). Also, in `.tex` files, the leftmost occurrence of a `%` marks the beginning of a comment that runs from this symbol included all the way to the end of the physical line, including the `\n` character. This allows one to have many physical lines to represent a unique logical line, and can be used to, for instance, have `\blackcases{_/_, ..., _/_}`, `\words[v]{_/_/_, ..., _/_/_}`, `\words[h]{_/_/_, ..., _/_/_}` and `\gridcross{_, ... _}` split over more than one physical line and improve readability. It is therefore advisable to process a `.tex` file in such a way that its content is captured as a single string with all space removed, before searching for patterns of interest in this string.

2.2 Analysing a grid (3 marks)

Your program will allow `Crossword` objects to be created from `.tex` files that you can assume are stored in the working directory, and whose contents satisfy all conditions spelled out in Section 2.1. Making use of the `.tex` files `empty_grid_1.tex`, `empty_grid_2.tex`, `empty_grid_3.tex`, `partial_grid_1.tex`, `partial_grid_2.tex` and `partial_grid_3.tex`, having for associated `.pdf` files empty_grid_1.pdf, empty_grid_2.pdf, empty_grid_3.pdf, partial_grid_1.pdf, partial_grid_2.pdf and partial_grid_3.pdf, here is a possible interaction.

Passing as argument to `print()` a denotation of a `Crossword` object results in an output of the kind

```
A grid of width _ and height _, with _ black case[s], filled with _ letter[s],
with _ complete vertical word[s] and _ complete horizontal word[s].
```

where

- the first occurrence of `_` is a number that represents the horizontal dimension of the grid,

- the second occurrence of `_` is a number that represents the vertical dimension of the grid,
- the third occurrence of `_` is `no`, `1` or a number at least equal to 2 that represents the number n of black cases in the grid (of course it is `case` if $n = 1$ and `cases` otherwise),
- the fourth occurrence of `_` is `no`, `1` or a number at least equal to 2 that represents the number n of letters alredy placed in the grid (of course it is `letter` if $n = 1$ and `letters` otherwise),
- the fifth occurrence of `_` is `no`, `1` or a number at least equal to 2 that represents the number n of vertical words in the grid (of course it is `word` if $n = 1$ and `words` otherwise), and
- the sixth occurrence of `_` is `no`, `1` or a number at least equal to 2 that represents the number n of horizontal words in the grid (of course it is `word` if $n = 1$ and `words` otherwise).

2.3 Filling in a grid with given words (5 marks)

`Crossword` objects can call the `fill_with_given_words()` method, that takes two arguments.

- First, the name of a `.txt` file, meant to exist in the working directory and store a number of words, one word per line (without empty line, without space on any line). You can assume that the file indeed exists and has the expected contents. The file could be empty, and there could be more than one occurrence of a given word.
- Second, the name of a `.tex` file, meant to store a representation of a solution in case a solution exists.

The aim is to complete all missing entries precisely with the words stored in the file provided as first argument; so there should be as many words in the file as there are incomplete words in the grid; words do not have to be distinct in the file, because a given word can appear more than once in a grid.

If the task is impossible, then the method prints out:

```
Hey, it can't be filled with these words!
```

Otherwise, the method prints out:

```
I filled it!
Result captured in _.
```

where `_` is the name of the `.tex` file that has been provided as argument to the method. The solution that has been discovered is stored in that file. If the file does not exist then it is created, in the working directory; if it does exist then it is overwritten. There could be more than one solution; only one solution is sought after, and any correct solution is acceptable. In any case, the method returns `None`.

Here is a possible interaction.

- The `.tex` files that have been created or overwritten during this interaction are
  - `filled_empty_grid_2.tex`,
  - `filled_empty_grid_3.tex` and
  - `filled_partial_grid_2.tex`.
- The associated `.pdf` files are
  - filled_empty_grid_2.pdf,
  - filled_empty_grid_3.pdf and
  - filled_partial_grid_2.pdf.

It is advisable to make sure that the spaces in the `.tex` files produced during the interaction are exactly as shown. Still, whitespace will be ignored for assessment purposes, but of course, all other nonspace characters have to be exactly the same, character for character.

2.4 Solving a crossword (5 marks)

Crossword objects can call the solve() method, that takes as argument the name of a .tex file, meant to store a representation of a solution in case a solution exists.

The aim is to complete all missing entries with words from the file dictionary.txt. That file is provided, meant to be stored in the working directory, and you can assume that this is the case indeed. A word in dictionary.txt could be used more than once, because a given word can appear more than once in a grid. The grid could already contain complete words that are not in dictionary.txt. This is fine. It should **not** be checked that any complete word in the input grid is in dictionary.txt; this might not be the case, and that is fine. But any incomplete or missing word has to be one from dictionary.txt.

If the task is impossible, then the method prints out:

```
Hey, it can't be solved!
```

Otherwise, the method prints out:

```
I solved it!
Result captured in _.
```

where _ is the name of the .tex file that has been provided as argument to the method. The solution that has been discovered is stored in that file. If the file does not exist then it is created, in the working directory; if it does exist then it is overwritten. There could be more than one solution; only one solution is sought after, and any correct solution is acceptable. In any case, the method returns None.

Here is a possible interaction.

- The .tex files that have been created or overwritten during this interaction are
  - solved_empty_grid_1.tex,
  - solved_empty_grid_2.tex,
  - solved_partial_grid_1.tex and
  - solved_partial_grid_3.tex.
- The associated .pdf files are
  - solved_empty_grid_1.pdf,
  - solved_empty_grid_2.pdf,
  - solved_partial_grid_1.pdf and
  - solved_partial_grid_3.pdf.

It is advisable to make sure that the spaces in the .tex files produced during the interaction are exactly as shown. Still, whitespace will be ignored for assessment purposes, but of course, all other nonspace characters have to be exactly the same, character for character.