The deadline for this exercise sheet is **Tuesday, 29.05.2018, 14:00.**

Introductory Words

This week we decided to put the hints and structural ideas as well as the pseudo code into another file. This way you can challenge yourself and only peek bit for bit when you are stuck. The hints are structured with a table of contents and only one hint per page. This way you can very selectively look at what you are struggling with.
Feel free to let us know your thoughts on this!

1 A Way Out

In this week's homework you will build a program that can find the way through a maze from a given start point to a given goal point using backtracking.
In the .zip archive, you will find a folder called *mazes*. Inside are several text files each containing one maze. A maze is a grid structure, similar to the tic-tac-toe game, each cell containing one character, which represents a different part of the maze. We chose:

- `#`: to represent the hedge / wall of a maze, so a field that is unpassable.
- `*`: to mark the start field, so the cell from which we start to find a path.
- `G`: to mark the goal cell, which we want to reach.
- ` `: A space that represents the path that we can walk on.

An example might be:

```
##########
#        #
# # # #  #
# # ### ##
# ###*#  #
# #   ## #
# # #### #
G #      #
##########
```

Your program now should be able to find a way from `*` to `G` and show the taken path. Diagonal movement is not valid, and you can only visit the 4-neighbourhood (the up, left, down, and right neigbhour cells). Use the trial-and-error strategy of backtracking to force your way through the maze. Your program should be able to deal with mazes that have no valid path or no start or goal pos. Split your code into modules such that your code can be combined under meaningful modulenames and document all of it appropriately. Don't forget the module docstring! (see the hangman solution for an example)
Small-ish Bonus Task:
To dynamically check the folder for all available files, have a look at the `os` module `https://docs.python.org/3/library/os.html`, and esepcially at the

`listdir(path)` function. Together with the `open` function, this should allow to read all mazes, present them as a choice to the user, and then open the chosen maze. *End of small-ish Bonus Task.*
Small-ish Bonus Task 2:
Let the user input a new maze in the terminal, and save it in the folder with the other mazes. *End of small-ish Bonus Task 2.*

1.1 Bonus: Bundle it up!

Note: This will involve a bit of reading and maybe some web searches from yourself, so it is hard to say how much time this task will need. It also depends of course on how far you want to take this.

When you have programmed your amazing maze solver (you choose whether the maze or the solver is amazing), you have programmed several (maybe around three?) modules. These modules together build a unit to solve a maze. So let's bundle them up and build a package!
The modules should reside alone inside their own folder, the folder name is your package name. A package needs a `__init__.py` and a `__about__.py` (technically it only needs the init file).
The `__init__.py` file tells Python that this directory is a package. The file can even be empty, just its existence is enough. However, the `__init__.py` file can be used to initialise certain code or define certain package relevant variables. You can find an introduction to defining packages here `https://docs.python.org/3/tutorial/modules.html#packages` and as well how to import modules inside the same package.
For the `__about__.py` file you might want to look at this blog `http://toxi.nu/blog/how-to-store-your-python-package-metadata/`, which gives a short rundown on why you would want to use the about file and what to put in it.

If you want to know more on packages and how to prepare a package for release to be used by third parties, you can have a look at `https://packaging.python.org/tutorials/packaging-projects/`.