Documentation 🛰️

Introduction

A strategic terminal game based on Minesweeper from 1990. Reveal all tiles on a grid while avoiding the bomb-infested ones.

Features

Abandoned Space Station makes use of the general game-play from *Minesweeper*.

- A grid (*default: 7x7*) with random bomb placement (*default: 10*) is generated. Tiles with no bombs contain a digit indicating how many bombs surround this tile (adjacent tiles)
- The player can reveal those tiles. Uncovering a tile with a bomb means game over.

Other functions are:

- The player can move his cursor around using the keyboard. Actions such as *marking* or *scanning* a tile are performed with the selected tile.
- Most Minesweeper-like games also include a marking function. This is also possible here. With `X`, the selected tile is marked in red (`X` again to remove the marking).
- Each tile represents a room on a space station. When generating a new grid, a random name from `data/room_names.txt` is assigned to each room. Each name is used only once. If all names have been used at least once, a default is used for the remaining rooms (the last entry in the text file will be the default if not empty; *default: 'Hallways'*). This will lead to a unique map generation each round.

To increase the overall experience and make the game feel more alive, there are several feedback messages and story texts implemented into the game play:

- Before starting the game, the player can read an introduction text about his mission on the space station. The needed controls to play the game are then always shown next to the playing field.
- Besides the indicator being shown when revealing a room, there is also a feedback text about the current scan. Based on the situation, this will be a different text. Rooms with no bombs around show other messages than the ones being surrounded by bombs.
- Both winning the game by revealing all safe tiles or losing by scanning an infested tile lead to an ending text about how the mission went.

All story elements come in different variations and are randomly chosen when displayed. They can all be found in `data/room_names.txt`

Libraries

Only the default packages have been used during development. Besides my custom modules, the following third-party modules are implemented:

- io
- os
- sys
- random
- unittest

These are the used libraries (also listed in the `requirements.txt`):

- Coverage 7.6.12
- MyPy 1.15.0
- Pylint 3.3.4

Architectural Description

The project is generally structured as follows:

- `root/`
  contains the whole project. This is where we run the game from for instance.
- `data/`
  the .txt-files in here contain all story texts and messages.
- `src/`
  contains all game files and modules.
- `tests/`
  contains all test files using *unittest*.

Let's look deeper into the specific Python files:

`src/class_room.py`

This file contains the *Room* class. Each room or tile on the playing field is represented by an object of this class.

`src/file_reading.py`

This file handles file imports from the data folder. Texts and messages are read from file into the program to be used while playing.

`src/main.py`

This is the brain, the engine of the game. When running the game, this file is being run. All other functions in external scripts are called from here.

`src/player_actions.py`

Whatever the player's inputs are, they are handled right here. This includes moving the cursor, scanning or marking a room, and typing any other unintended bu****it into the console.

`src/playing_field.py`

Maybe the most important script besides the main-file. This file contains all relevant functions for the game board, whether it's generating it before the game starts or writing it to the console. Out of all scripts, it holds the most lines of code.

`src/utils/constants.py`

I really wanted to have a dedicated place for all constant variables. The grid size, bomb count, controls and all other set values are in here and imported in nearly every other module.

`src/utils/error_handling.py`

Since I didn't need many custom exceptions, only the *InvalidActionError* (raised when player input is unknown) lives inside this file.

`src/utils/text_formats.py`

This file contains helper functions to return formatted text snippets. There is also a clear-function to wipe the terminal empty.

`test/helpers.py`

The test functions don't really need their own explanations. This file though is not directly a test script. It rather holds functions used in more than one test script to make them reusable.

User Interface

The UI must be user friendly and look good since this is what the player sees during the whole game. The Idea was to make it look like a real terminal that shows up on a portable machine carried around during the mission. It shows a grid with unknown tile contents at first. Blocky and straight symbols make it look more retro.

Next to the playing grid, there is an info area showing the objective and the game controls all the time. At first it doesn't look outstanding, but it was not easy at all to correctly place the text there since terminals can only print text from left to right on the current line.
With some calculations on the size of the grid, it was possible to make it dynamic. It will keep its' general look even if changing the size of the grid or the info text itself. Making the text bigger than the grid itself will not work correctly.
Based on the keys set as controls in `constants.py` the overlay will show the correct ones.

Under the playing field, we can see the input area. It will always show the current position of the player. There may be story-related messages being shown in this region if a scan has been made in the last turn.

In terms of coloring, the main colors are yellow and orange. The grid itself shows basic colors which should be self-explanatory for humans (green is good, orange is a warning, red means danger, and so on...)

Program Flow

The player starts the game by changing the terminal directory to root and running `./mission_start.sh` (.bat for Windows).
=> *The shell script runs the `main.py` file which starts the game.*

The introduction shows up, telling the player about what happened and why he is here. It is a mission to safe the Sevastopol space station. By pressing any button, the game starts and the player now sees the main user interface. It becomes clear that the grayed tile indicates the player selection. Next to the playing field, the player is informed about how many bombs there are in total and what controls he must use to play the game.
=> *The main function called the introduction and grid generation. The game is now waiting for user input indefinitely.*

An input event is waiting for the player to type something into the console. At first, he is testing if moving with `WASD` even works. The selection moves through the rooms.
Before risking to instantly lose the game, the player tries to mark one of the tiles. A hazard symbol is shown. Marking again removes it.
=> *User inputs are transferred to another function. If the input is valid, the grid updates its' state accordingly.*

Finally, the player gathers the courage to scan a room on the playing field. An orange warning symbol with the digit `2` is shown. A message states that it is safe in here, but there are dangers nearby. It is now clear what the player is supposed to do. By scanning other rooms, the player is able to determine rooms which must contain bombs. While marking and avoiding such tiles, the player is able determine the position of all bombs. Scanning the last safe tile leads to the player winning the game. A message congratulates him for his efforts.
=> *Each turn, the game analyzes the grid to check if the winning condition is met. If this is the case, the while-loop is exited.*

In another game, the player was unlucky. Without any determinable tiles or enough information, the player scans an infested tile, leading to a game over. All infested tiles are revealed, showing where the bombs would have been located. By pressing enter, the terminal is closed.
=> *If a scanned tile turns out to contain a bomb, the while-loop is instantly exited with the game over condition. The player loses, no matter his progress.*

Since the grid is generated randomly each round, every game will look different than the ones before.

Test Results

Coverage

By running `./coverage.sh` in the root directory, all unit tests are being run and analyzed, resulting in an html-file showing the results:

File	Statements	Missing	Excluded	Coverage
src/__init__.py	0	0	0	100%
src/class_room.py	30	0	0	100%
src/file_reading.py	19	0	0	100%
src/main.py	89	0	2	100%
src/player_actions.py	24	0	0	100%
src/playing_field.py	114	0	0	100%
src/utils/__init__.py	0	0	0	100%
src/utils/constants.py	15	0	0	100%
src/utils/error_handling.py	11	0	0	100%
src/utils/text_formats.py	11	0	0	100%
tests/helpers.py	16	0	0	100%
tests/test_class_room.py	6	0	0	100%
tests/test_file_reading.py	10	0	0	100%
tests/test_main.py	59	0	0	100%
tests/test_player_actions.py	18	0	0	100%
tests/test_playing_field.py	18	0	0	100%
tests/test_text_formats.py	12	0	0	100%
Total	452	0	2	100%

After a lot of research, I haven't found a way to test the following part:

```
# main.py

if __name__ == "__main__": # pragma: no cover
  main()
```

To make a test coverage of 100% even possible, I decided to exclude this part from being tested by adding `# pragma: no cover` to the end.

Some scripts didn't need their own unit tests since Coverage already showed 100% for them.

Pylint

By running `./pylint_test.sh` from the root directory, the Pylint tool performs tests on all scripts inside the src and tests folder. Both return a score of 10.0/10.0 which means that Pylint can not find any error in the code. The Pylint extension for Visual Studio Code also doesn't highlight any part of the code.

MyPy

By running `mypy ./src` and `mypy ./tests` from the root directory, we get a green result message stating that there have been no issues found in the source files. MyPy does not find any error nor does the VSC extension show any.

It total, all test tools return perfect results. The program is built as intended and fully stable.

Made by Dominic Brauer
Documentation created in Markdown and exported as PDF. Structure and appearance may differ.