Week 5. Generating Code

Arsenii Stratiuk, Roman Leshchuk, Kylle Basillo

First, before the meeting, we wrote the code ourselves, without AI tools. To do this, we had to create three functions that were specified in the condition, as well as two additional ones: main to call the written functionality and compare_rating for correct sorting in the top_n function. When writing, we adhered to the PEP-8 style and tried to anticipate all possible cases of input data, for this we wrote doctests. Furthermore, we used Git and GitHub to collaborate on the code and control the versions. Our repository can be found here: https://github.com/ArseniiStratiuk/Film-Search.

In particular, in the top_n function, it was important to check the correctness of the output when several genres were specified, the number 0 instead of the size of the output array, which meant including all elements, as well as the speed of operation for large input data, in particular, it was necessary to make sure that the algorithm works in O(N), because the list length could potentially be more than 1000. A separate subtask was to develop a key function for sorting that takes into account not only the movie rating, but also its lexicographical size of its title in case of rating matches.

In the read_file function, it was necessary not only to read the file, but also to make sure that the data was written correctly. In addition, it was first necessary to check whether such a file exists and whether the year for filtering films met the conditions.

The write_file function also includes numerous checks, such as a deep check of the input data types. It was also important to check that the file format to which the writing would be carried out would be .txt. Only after that the data could be written to the file, checking for possible errors during writing.

Having written the code, we made sure that we had not missed anything. At this stage, checks were also added for various edge cases, such as passing the wrong types of arguments to the function or other situations that could cause the code to execute incorrectly. A larger number of doctests were also added for proper verification as well as the general logic of the use case scenario to use the scripts functionality.

To generate the AI version of the code, we have used the extension of the ChatGPT model specifically tailored for programming tasks. This is the exact prompt that was asked: https://chatgpt.com/share/67338a5d-bb64-8009-802c-e0d44fa27208. We used strategies such as giving step-by-step instructions that were provided with the assignment, assigning the model the persona of an experienced programmer, dividing the prompt into parts, and specifying that we needed an expert-level, self-documented and clever code which would align with the PEP8 standards. I also clarified what tools were not allowed, specifically any imported libraries, lambda functions or generators, which are all common to solve this type of problem, and ChatGPT tended to use them before specifying clearly what we needed.

The code was correct and aligned with the assignment requirements from the first try with few exceptions. It had the same decomposition structure and mostly followed all PEP8 rules like we did. The only nuances that were highlighted by Pylint are missing module docstring that was commented instead of being in the docstring, and also this message *"Consider explicitly re-raising using 'except IOError as exc' and 'raise IOError(f'An error occurred while writing to the file {file_name}.') from exc'PylintW0707:raise-missing-from"* the explanation of which can be found here: https://pylint.readthedocs.io/en/latest/user_guide/messages/warning/raise-missing-from.html - *"Python's exception chaining shows the traceback of the current exception, but also of the original exception. When you raise a new exception after another exception was caught, it's likely that the second exception is a friendly re-wrapping of the first exception. In such cases, `raise from` provides a better link between the two tracebacks in the final error."* However, in general the code was completely valid and completed the task, although being slightly different from ours. In instance, we did not receive the mentioned Pylint message because of just returning None when getting an error instead of raising an exception, which can be useful for using the function as an imported one. One of the more questionable choices of the AI model was to skip the invalid lines of input data instead of stopping the function's work. In such case, if part of the data is correct, it is better to return None and message that the data is incorrect. Partial data reading can cause problems in the future, which are hard to track. Also, the first version included a lambda function, which was not allowed according to the prompt. Thus, we had to regenerate the code one more time, finally getting the final code. In the end, we have fixed all of its formatting problems and fine-tuned doctests that we used, because the implementation of the top_n() function differed, giving slightly biased results – our streamlined approach makes it less prone to rounding or floating-point discrepancies, as the calculations are reduced to fewer steps. More specifically, among the differences were: we use a dictionary (actors_max) to store each actor's maximum rating across all movies in data. This way, it only calculates each actor's maximum rating once, which is more efficient; After collecting each actor's max rating in actors_max, our code looks up each actor's max rating and calculates the average directly. This is efficient since each movie's Actors_Rating can be computed in constant time by

accessing actors_max. Also, we differed a bit in error handling approaches, checking more input to be invalid. Overall, we got a decent version of the solution created by ChatGPT in short time that passed all of our doctests.