Kathrin Betzenbichler
Matr.-Nr. 796312

Internship report

1. Project description and approach

The goal of my project was to patch all the project files of the Wintersemester 2018/19 "Linguistic gaming with python" course together and therefore make them playable as one whole consecutive game. In order to do so, I not only had to figure out how to make one game out of thirteen project files, I also had to fix certain issues of certain programs, for example problems with robustness. Although the focus lied on that, I wanted to reduce repetitiveness of the minigame files and working on setting up functions, where necessary, as well. To make one game out of thirteen different games, I decided to set up a main game file. Instead of just copying and pasting the code of every group inside the main game file, I call up all the minigames as separate scripts using the import statement. This way, the project programs (minigames) remained separate files and the main game file would not get too overwhelmingly big. As a basis for the main file I used the "opening&final.py" program file of group 1 (Price/Ma/Youn). In order to keep the file as clear and comprehensible (which is usually short) as possible, I also exported the opening description function to a separate file ("OpeningDescription.py") and import it to the main game via an import statement at the right place. Every minigame file was renamed to whatever objective the group had, e.g. providing information about a case (like allative) or a specific word (like "cheyao") in the Dothraki language the groups were working with. Starting the main game file starts the whole game and allows to play all minigames in succession.

2. Changes regarding the submitted group project files

2.1 Main game file "game.py"

The main game file consists primarily of group 1's project file, which I renamed to "game.py". They set up a *Hero* class object, which is a good solution for storing and calling up information related to the hero of the game. However, the class did undergo a few changes:

I added a "scripts" list, to discern between items (inventory) and linguistic information (scripts). Also, I came up with functions for showing, adding or removing items to both, inventory and scripts. Besides that, I introduced another function *check()* for checking if the health of the hero has fallen to or below zero. If so, the health is restored to 100. This is an important addition I made to be able to play the whole game with all of the minigames through. Since some of the

minigames decrease the health, it could fall to or below zero, which would then likely lead to an error at some point, because some of the groups/programs use health as a condition for while-loops. Therefore, I added the *hero.check()* function and call it up in the minigames when it is necessary.

In the main game program, the hero is instantiated after the opening description ran. To be able to pass health, inventory and script information from one file to another (which means to the minigames and also back to the main game file again) and also to use the functions of the hero object within another file (the minigames), I had to come up with a solution. The easiest option seemed to be to store the hero object in a binary file by using the pickle module. The file can then be opened and used, and also permanently changed, in the minigame files.

2.2 Minigame "Accusative.py"

Group 2 (Osime/Hotoboc/Müller) came up with three small challenges, in which the player has to guess the accusative case. I changed the overall functionality to having a dictionary with the nominative as keys and corresponding accusative forms as values, instead of separate lists or even creating the accusative manually by stripping or adding characters from the string, like the group had before.

Furthermore, I implemented a *wordChoice()* function to get rid of repetitive code (choosing a word manually every time) and to make sure that only words which have not been used before are getting chosen. Following this change, I smoothed out the code while making use of the dictionary (e.g. deleted unnecessary variables) and fixed issues like robustness (handling false input) by implementing while-loops.

In the first challenge function *writeAcc()* there were some lines I could simply delete because of the dictionary solution. In the second challenge *chooseAcc()* I simplified the code for coming up with the fake accusatives and printing them. Because the code for scrambling up a word in the third challenge *scrambledAcc()* didn't function properly, I changed it by using the random.shuffle function to create a jumbled version of the chosen word.

2.3 Minigame "InfinitiveMorpheme.py"

This minigame by group 3 (Schweizer/Willenborg/Hofenbitzer) consist of a main game file and three other files (minigames), which are imported to the main file. The only changes I made to the main game file was to comment out the pygame princess animation in the end, since it was not working properly and caused the game to crash. The minigames were changed as follows:

2.3.1 Minigame “mam.py”

I only fixed some minor issue with printing out the keys of the dictionary.

2.3.2 Minigame “fws.py”

In this minigame, the main while-loop was terminated after choosing 0, but this was not stated anywhere, so I added an explicit “exit” option for 0. The restoration of health when giving a right answer by 14 was changed to 7, because it seemed to unbalance the game. Other than that, I only added an else-statement to handle invalid input.

2.3.3 Minigame “hangman_ling.py”

The “hangman_ling.py” minigame was not changed at all.

2.4 Minigame “daki_davveya.py”

Group 4 (Glanville/Gicquel/Roshankar/Sierra/Oliynyk) decided to each program a minigame with a different setting. Since their resulting project file was very big and there was no need to have all of the five minigames in one file, I split them up into separate files and set up a main game file “daki_davveya.py”, which calls up the minigames one after another via import. It was not necessary to load the binary file with the hero in every minigame, since there is no change to health, inventory or scripts sometimes.

2.4.1 Minigame “ocean.py”

For the initial choices of weapons and distance I set up a big while-loop and a list with the weapons to choose from. The list is displayed with the index of each entry. The index is then used for making the choice. This is a more robust way to display and handle the input. Before, the chosen weapon was just caught as string input, without checking if it even is a valid choice.

In order to avoid errors like ValueError or IndexError while making choices I added try-statements. Another while-loop and if-clause is securing the second choice, which has to be an integer. The following if-elif-clauses were initially set up for displaying the winning message. Since all of the conditions led to the same message, I put them together into only one if-statement, connected with the “or” operator.

For the mermaid section, where the player has to choose one of three options for three consecutive scenarios, I changed the mechanics to having only one while-loop for every scenario with different if-elif-statements, which handle the input choice (choosing an option). Before, every “incorrect” option (input) led to a while-loop. The problem with this was that there was also a while-loop for invalid input (input which is neither “a”, “b” or “c”, so none of the available options), which had the condition “while answer != correct option”. That meant,

once the player typed in anything else than the correct option, the program entered this while-loop and then it could only be left with the correct answer. So, if the player answers with valid input but the wrong option, the while-loop for invalid input was entered instead of the while-loop for the corresponding incorrect answer. To avoid this, I switched to having one while-loop per scenario with different if-elif-statements for the available options. The while-loop is still terminated only if you answer correctly (break statement inside the if-clause for the correct option). But if an incorrect answer is given, the corresponding output is now displayed (chosen by one of the elif-clauses), the player is asked for input again and the while-loop iterates anew. Invalid input is handled via the else-statement, which also asks for new input and iterates the while-loop again.

2.4.2 Minigame “jungle.py”

In this game, the player has to answer three questions out of five correctly. I adjusted the if-clause, which displays a message after getting the fourth answer wrong, to “if right <=1” instead of “if right==0”. Furthermore, I copied the if-clause to appear after getting the third answer wrong with the condition “if right < 1”. If the player loses, the hero’s health is reduced by 5 now. This means, the binary file with the stored hero object is loaded in the beginning of the program.

2.4.3 Minigame “temple.py”

The goal of this game is to guess a certain word. I changed the condition of the while-loop from having a “tries” counter to the condition: while the answer is not “silence” (which means, not right). Therefore, I completely deleted the tries variable and shortened the if-else-statements within the while-loop to simply break the loop if the answer is “silence”.

2.4.4 Minigame “volcano.py”

The setting in this game is: the hero is playing a dice game with the volcano and can either believe or not believe the result the volcano tells the hero. I fixed the if-statement for deciding if you believe or not and changed it to a whole if-elif-else-clause. The else-statement catches invalid input now. Also, I implemented two while-loops to allow to loop over again when the else-statement is executed because the input given by the player was invalid.

2.4.5 Minigame “desert.py”

In this minigame I had nothing to add, except adding the translation for “daki” and “davveya” to the hero’s scripts in the end as the reward for playing.

2.5 Minigame “addakhat_azhat.py”

Once again, I added nothing except the meaning of “addakhat” and “azhat” to the hero’s scripts after beating the minigame, programmed by group 5 (Gritzo/Kurch/Meurer/Purpus).

2.6 Minigame “Rosemary.py”

Group 6 (Nusser/Morgantti) invented a graphical minigame using tkinter. The game was working well, so I had nothing to add here. The only change I made to the code was that after successfully playing the game, “rosemary” is added to the hero’s inventory and “dave = rosemary bush” to the scripts.

2.7 Minigame “Pomegranate.py”

This minigame by group 7 (Xiao/Li/Gao) starts with a small animated pygame game, where the player moves a character. In the responsible *runGame()* function, I added coordinations as conditions to the character movement if-clauses in order to prevent the player from moving out of the displayed window. Furthermore, I fine-tuned the coordinates for finding the right tree, which was the goal. To make sure there is a small time delay after the user finds the pomegranate tree, I added the pygame.time.wait function. This way it displays the message that the right tree was found for a few seconds before continuing with the introduction. Before my fix, the message was not visible.

After the introductory minigame with the pomegranate tree, there’s an introduction text and a menu with different buttons displayed. Here, I fixed the bug that the program crashes when the player clicks on “achievements” before playing the following word choice game for the first time. This was done by introducing “achievement1” and “achievement2” as global variables right at the beginning of the program and assigning zero to them. This way, it simply displays a message that you have no achievements yet. Inside the *question1()* and *question2()* functions (which are responsible for the word choice game) I had to call them up as global variables, to be able to change them, since the hero should get an achievement for every right answer.

The group set up the word choice game so that the player could replay. While working on the game I realized, that at the beginning of each question, the corresponding achievement should be set to zero, so the user could replay without getting too many pomegranates as achievements. Since the initial game would only display one or two pomegranates as achievements, if you replayed and got more, there was nothing displayed anymore, but it was still counted in the achievement variable. Because the idea was to get only one or two pomegranates as achievements for the game, I added the *pomegranate_replay()* function, which is displayed

after playing the word choice game for the first time, instead of simply going back to the menu. The *pomegranate_replay()* function is essentially a copy of the introduction menu and displays just the same, except for the text. The text now warns you that you lose your current achievements if you play again. Since I couldn't edit the original graphic where the introductory text was printed on, I created a new one with Microsoft Paint as a quick solution for the replay text.

A small detail to fix was found in the *final()* function, where I put the for-loop for mousemotion outside of the if-else-statement. Before, the for-loop was inside the if-else-statement and therefore doubled, although it should run regardless of which statement is being entered.

While patching all the games together I noticed that the *terminate()* function not only terminates the "Pomegranate.py" game, but also the whole "game.py" program. Unfortunately, I couldn't just delete the system-specific sys.exit() function to fix this, since the program would give me an error that the fonts cannot be loaded anymore. Most likely, this is due to the while-loop, which was not interrupted, while pygame was quit already. Therefore, I decided that instead of exiting sys and pygame as the exit move, I should implement the termination of the current while-loops first. Therefore, all of the while-loops in every function were changed from "while True" to having a variable as a condition. The variables are set to "True" in the beginning of the program. Once the player wishes to exit and presses the "Quit" button, the *quitmenu()* function is being called up. I altered the function so that once it gets called, all of the conditional variables for the while-loops are set to "False", so the while-loops would terminate and therefore the game would end without exiting pygame or sys. Pygame is being closed at the very bottom of the program in the main scope after exiting all the while-loops by calling pygame.quit(). This closes the game window. Since the while-loops are already exited by then, there is no error anymore and the main game file keeps running.

To be able to put the achievements, which are pomegranates, to the hero's inventory for later use, I added a function *inventory()* to the program. The function is called up in the main scope after the game ends. It consists of opening the pickled file for the hero, checking for achievements with the *get()* function and a for-loop, which adds the string "pomegranate" for every element in range(achievement) to the hero's inventory. Finally, the inventory is printed out with the *hero.show_inventory()* method and the file is closed.

2.8 Minigame “akat.py”

The minigame around the word “akat” by group 8 (Baumann/Gustedt) is constructed as something similar to a maze. The player has six doors to choose. Opening the doors trigger a different scenario or event. The majority of the code was working well, so I didn’t have to change much. Only the code regarding door C and D was adjusted. In the code belonging to door C, I fixed an issue with generating a jumbled version of a word by using the random.shuffle function, like in the “Accusative.py” minigame. Also, I deleted some unnecessary code.

The code for door D is set up as a battle sequence. I fixed the malfunctioning try-statements and while-loops for item choice and the battle by adding the right conditions. The random enemy choice was adjusted as well. Implementing another try-statement should prevent the program to crash from an IndexError after all enemies are fought. Because there is the chance that the hero loses a lot of health during the battle, I added the function that if health falls below zero, it is restored to half, so it is possible to keep playing.

After experiencing every scenario, the script for “akat” is added to the script inventory of the hero.

2.9 Minigame “dorvof.py”

Group 9 (Reichmann/Hohl/Do) created three separate minigames in separate files, which are imported into a main game file. The idea was that the player can choose between two options and each option should trigger another game. At the end of either game, the third minigame (“Lava-Schollen-Spiel.py”) was imported and after beating this game, the main game completes. Since one of the minigames (“sprite_collect_blocks_levels.py”) wasn’t functioning, I changed their main game frame to either import the minigame “trivia.py” or the minigame “Lava-Schollen-Spiel.py”, instead of importing “trivia.py” or “sprite_collect_blocks_levels.py”.

Unfortunately, the “trivia.py” minigame arose the same problem as the Pomegranate game: The sys.exit() function for quitting the game terminated the whole Python process. Once again, I deleted sys.exit() and pygame.quit() and implemented a “running” variable as condition for the the main while-loop instead, which is set to “False” if the player either quits or successfully plays the game. After escaping the main while-loop, pygame can be terminated with the pygame.quit() function and the second minigame, “Lava-Schollen-Spiel.py”, is being imported, just as it originally was thought out. To make everything coherent, I added some flavor text to both minigames. At the end of the “Lava-Schollen-Spiel.py” the player finds out what “dorvof” means and the hero, which is loaded from the binary file, gets a script for the script inventory.

Furthermore, I added that if the player loses the minigame and has to play again, the hero is losing 5 health points.

2.10 Minigame "cheyao.py"

This minigame is set up as a question answering game, developed by group 10 (Wiesner/Nur Gül/Nguyen). They implemented two "jokers" the player could use if he/she doesn't know the answer. However, the program showed the bug that if the player chooses to use a joker but the "joker" list is empty because they have been used, it jumps to the next question without letting the user answer. This was fixed after changing the overall setup to having a function, but more importantly, a while-loop for the joker-choice. The while-loop had the following condition: while list with jokers is not empty. If it is empty, it would simply jump to the next step, which is picking an answer.

As mentioned, to reduce the amount of code and repetition, I created a function *questions()* which prints the questions, handles the joker-choice and checks the given answer. The function has two parameters (question, correct answer). Inside the function I put a print-statement for the corresponding question using the parameter, the while-loop for the Joker choice, asking for input ("answer") and an if-else-statements for giving output depending on giving a correct or wrong answer. Due to setting up a function, there were some changes I had to make to the code if the 50:50-Joker is chosen. I introduced a list with possible answers (a, b, c, d), from which I remove the correct answer. After that, I let one element get randomly picked from the list to be the "fake" option, which is presented to the user. The correct and fake answer are put into another list ("possible"), which is then sorted to be presented to the user in alphabetical order with a print statement finally.

Outside of the function in the main scope, I set up all of the questions and correct answers as variables ("question1" to "question10", "correct1" to "correct10") and introduced another variable "remainingQ" at the top of the code as a counter for how many questions remain. The variable "remainingQ" is reduced by one after every question that has been asked as part of the function *questions()*. In the main game scope, after the function and the variables for questions and correct answers, a while-loop follows (mostly to be able to play again after losing), with another while-loop in it, having the number of remaining questions ("remainingQ") as a condition to be more than zero (while remainingQ > 0). Inside this second while-loop I call up the function for every question with the corresponding question and correct answer given as parameters, one after another, from one to ten. After this, the inner while-loop is interrupted because the remaining questions variable is set to zero. Then the text for the end of the game is

being displayed with if-elif-else-statements. After successfully playing the game, information about “cheyao” is appended to the scripts list of the hero, which is loaded from the binary file. If the player fails and also loses all of the hero’s lives, the hero’s health is reduced by 10. If the player fails but doesn’t lose all of the hero’s lives, he simply gets a message. Finally, the player can choose to play again. Some minor problems, like replacing the name of the variable “health” with “lives” to avoid confusion and adding the .lower method to “wahl” variable for input, have been taken care of as well.

2.11 Minigame “Allative.py”

The “Allative.py” minigame by group 11 (Uygun/Beckert) only had to undergo some minor changes. I added a break statement to the if-clause for winning to be able to exit the game, and deleted the unnecessary resetting of variables/lists here. The “health” variable was altered to “hero.health” and the restoration of the health if the hero falls below zero to a gain of 50 instead of 100. Three variables were created for stating the allative rules. They are put in the scripts inventory if the player is successful.

2.12 Minigame “Imperative.py”

Changing the variable “health” to “hero.health” and “inventory” to “hero.inventory” was the only thing necessary to do for group 12’s minigame (Dryer/Lelner/Kuyrukcu). I decided to let the list of “sphynx_gifts” (which represent weapons) be added to the hero’s inventory. After playing, the user gets information about the imperative form for the hero’s scripts and the “sphynx_gifts” are removed from the inventory again, so that ideally, the hero only has rosemary and pomegranate in his inventory.

2.13 Word order

Unfortunately, I never received any functioning code from group 13 (Shahto/Volk) as the minigame regarding word order.

2.14 Last Level inside main file “game.py”

The last level was part of group 1’s work. For the last level, they set up two classes. I added the __init__ method to both, which automatically initializes the classes’ attributes.

The west and east scenario are two methods inside of the *Scenario* object class. I set up two dictionaries, “horses” for west and “goats” for east to make printing and choosing an option with if-elif-clauses easier. I also added a lot of else-statements for if-clauses to catch invalid input. Another addition made was that if the hero doesn’t have anything (which should be the pomegranates and rosemary) in the inventory from the previous games, the bad ending is being

triggered, because the hero can't get help from The Ibex and the Dark Bay Horse, which is the only way to beat the game ultimately. Also, if the player tries to feed the Dark Bay Horse but only has one pomegranate instead of two, he fails and gets the bad ending. After successfully feeding the horse or the ibex, I remove the corresponding items (two pomegranates or rosemary) from the inventory.

2.14 Bonus game after finishing the main game: "CaveRoomBonus.py"

Although the code for the so-called bonus level (Price) was very elaborate, it was too repetitive, so I decided to work on instantiating functions. The first part of the code is for drawing descriptions and the game surface. A lot of this can be done by re-using functions with parameters for drawing text (*drawText()*), pictures (*scaryEyes()*) and objects (*drawRect()*) on the screen, which I created. In order to draw the barriers on the surface, I made nested tuples with the coordinates, which are then iterated through with a for-loop using the *drawRect()* function.

After setting up all the drawing functions, I decided to improve the gameplay functionality as well. This started by changing the mechanics of the *monsterMovement()* function to storing the positions of the monsters in a dictionary instead of having a variable for each X-coordinate and Y-coordinate for every monster, so seeing as there are four monsters there were eight variables in total. In the new dictionary, the name ("M1" for monster 1, "M2" for monster 2 etc.) is functioning as the key, whereas the position of the monster is stored as a list representing the value. The X-coordinate is the first entry of the list, the Y-coordinate the second entry. This way, the program can iterate through the dictionary with a for-loop and conduct the movement-check (checking if the given movement is allowed), as well as the actual movement (picking a random direction and calculate new positions) and storing the new positions all within the for-loop. The *drawMonsters()* function I initially created had then to be changed again, in order to use the dictionary with a for-loop. It turned out to be a bit tricky to figure out where to call up the functions with which parameters , especially inside of the main *playGame()* function, but after some trials it turned out to be the more elegant solution overall. By making these changes, I was also able to reduce the file length from 925 to 526 lines.

3. Summary

While patching all of the program files together, I encountered and had to overcome several problems. The most difficult of them were certainly to figure out how to pass the values and method functions of the *Hero* class object from the main game file to the other files and back,

how to prevent the program from raising an error after removing the sys.exit() function from the minigames using pygame and how to make sure that the game would not generate any error message at all and terminate, so therefore be playable as a whole. The code that constituted the biggest challenge was definitively the “CaveRoomBonus.py” program, since there were a lot of technical changes involved by introducing functions to the code and storing coordinates in a dictionary. But seeing as I could reduce the code to almost half the size, it turns out to be the more practical and efficient option.