Computational Thinking

Computationeel Denken - T1ACD2E

KU LEUVEN

Exercise session 10 – Example exam

This exercise session is organized as an exam. You will start with some simpler programming tasks and will proceed afterwards with two more difficult tasks. You can earn 20 points in total: 8 for the simple tasks, and 6 for each difficult task. The distribution will be similar on the real exam.

Simpler programming tasks

Scenario

In this part you will solve some simpler programming tasks.

Start from the file *lab10_simple_tasks_start.py*.

A group of runners tracked how many kilometers they ran for a week. Now, they want to obtain some knowledge from this data.

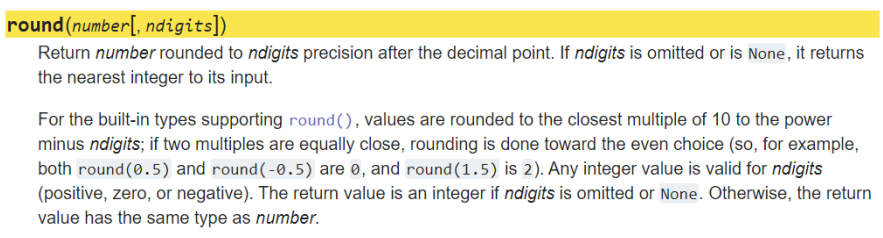
Task 1 [0.5 Point]: Give the number of runners as an integer who registered their kilometers. This group is presented in a list (*km_list)* and a parameter of the function. The list has the following format: [km1, km2, km3, km4, …].

Task 2 [0.5 Point]: Decide whether a runner is doing great based on the registered kilometers. The number of kilometers is an integer and parameter of the function (*km_value*). If the number is lower than 5 km, you return a string “not ready”, if the number is higher than 10 km; you return a string “good”. If the number is in between, you return a string “intermediate”.

Task 3 [1 Points]: Calculate the average number of kilometers of the group runners. The function has a list of the registered kilometers as a parameter (*km_list*).

Task 4 [1 Points]: Calculate how many runners ran a specific number of kilometers. The function has two parameters, a list of the registered kilometers (*km_list*) and the number of kilometers (km_value).

Task 5 [1 Points]: Write a function that converts a list of weekly total distances ran into a list of daily averages. The function has a list of the registered kilometers for one week as a parameter (*km_list*), and returns a list of daily averages, rounded to one decimal place. Note: the resulting list should only contain values larger than 1, if a daily average is lower, you should omit it from the list. Tip: use the *round* function:



round(*number*[, *ndigits*])

Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, it returns the nearest integer to its input.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are `0`, and `round(1.5)` is `2`). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise, the return value has the same type as *number*.

Task 6 [1 Point]: Write a function which checks if a certain distance is present in a file with kilometer values. This function takes two parameters: the name of the file (*filename*) and the value you are looking for (*km_value*). It returns True or False.

Task 7 [1.5 Points]: Write a function to grant a bonus when a runner is 50 years or older. In this case, the number of kilometers is increased with 5 km. The registered kilometers of other runners remain the same. The function has two parameters, the list of registered kilometers (*km_value*) and the ages (*age_list*) of the runners. Return a list with the updated values of the registered kilometers. If both parameter lists have different lengths, you should simply return the kilometer list.

Task 8 [1.5 Points]: Write a function to calculate the average age of all runners who received the feedback "good" in Task 2 and ran more than 10 km in a week. The function has a matrix as a parameter (*runner_matrix*) with a row per runner, containing the elements kilometers and age according to the following format: [[km1,age1],[km2,age2], [km3,age3],…].

Difficult task 1 - RPG

Start from the file *lab10_difficult_task1.py*. In many 2D role-playing games (RPGs), the playing field is visualized as a matrix on which both players and enemies can move. If player and enemy meet on the same field, combat takes place or the player takes damage; if the player has reached the limits of the playing field, moving further is prevented by the game.

5				
		2		
1			1	
				1

Figure 1. Example of a playing field with the player (5), weak enemies (1) and strong enemies (2).

In this task, you will implement functionality that allows the player to take steps on the playing field, and which then determines whether the player meets an enemy (resulting in immediate loss of health). The following rules apply:

- The player can only move in four directions: UP, DOWN, LEFT, and RIGHT.
- The player cannot move beyond the boundaries of the playing field. If this is attempted, no movement takes place.
- If the player encounters a weak enemy (1), this results in a loss of 50 health points (health).
- If the player encounters a strong enemy (2), this results in a loss of 100 health points (health).

1. **[1 Point]** Write a function **translate_move** that translates the direction into a 2D coordinate [a, b] depending on the string that was passed as a parameter (i.e., “UP”, “DOWN”, “LEFT”, or “RIGHT”). For example, “UP” would result in the same column (b=0), and a row above (a=-1). Note that the (0,0)-coordinate is positioned in the top left, with the positive direction of the y-axis pointing downward, which means you should lower the y-coordinate to move up on the screen.

2. **[2 Points]** Write a function **fight_enemy** that lets a player fight an enemy that’s positioned in the same spot, and returns the player’s resulting health. This function takes three parameters: the player’s health (*health*), the position on the playing field (*position*), and the overall state of the playing field (*world*) represented as a matrix. Note the player’s health can never drop below zero.

3. **[3 Points]** Write a function **player_move** that takes the same three parameters as the previous function, plus a fourth parameter *direction*, which represents the direction of the move (“UP”, “DOWN”, “LEFT”, “RIGHT”). The function determines whether the intended move of the player is possible, and what consequences it has for the player’s health. If the move is possible, the function returns a list that includes (1) the player’s health after the move, (2) the player’s position, and (3) the updated state of the world. If the move is impossible, the player does not move and nothing changes. Should the player die, they are no longer present in the matrix, since the position where they died will be occupied by the enemy that defeated them. In that case the returned position is the position where the player died.

Difficult task 2 – 5-7-9-rule

Start from the file *lab10_difficult_task2_start.py*. A course (OPO) at the university is a coherent set of learning and assessment activities (OLA). The final mark of an OPO is the weighted sum of the activity marks, converted to a grade out of 20.

This task starts with a 2D list with the results per student of all activities in an OPO for a group of students. The 2D list has the following format: [[student1_activity1, student1_activity2, student1_ activity 3, …], [student2_ activity1, student2_ activity2, student2_ activity3, …], [student3_ activity1, student3_ activity2, student3_ activity3, …],…].

You also have a second list with the weights per activity in the following format: [weight1, weight2, weight3, …]

1. **[2 Points]** Write a function **opo_score_without579** that returns a list with the OPO final mark per student based on the weighted activity marks. You can use the following formula to calculate the final mark:

   $$\text{Final\_mark} = \frac{((\text{activity1}*\text{weight1})+(\text{activity2}*\text{weight2})+(\text{activity3}*\text{weight3})+\ldots)}{(\text{weight1}+\text{weight2}+\text{weight3}+\ldots)}$$

   Note that a number with a decimal part lower than 0.5 is rounded down to the nearest integer, while a grade with a decimal part equal or higher than 0.5 is rounded up to the next integer.

2. **[2 Points]** Write a function **ola_scores_from_file** that reads a 2D list of OLA-scores from a file. The resulting list should be structured as indicated above. The function takes two parameters: the file name and a list of weights for all OLAss.

   All OLA-scores are stored sequentially in the file, grouped per student. For example, for a course consisting of 3 OLAs, the first three scores in the file belong to student 1, the next three scores belong to student 3, and so on. The amount of OLAs can be deduced from the length of the list of weights.

3. **[2 Points]** At last, write a function **final_opo_score** to apply the 5/7/9 rule. This function has two parameters, (1) a list of OLA-scores, structured as described above, and (2) a list of weights per OLA.

   The following rules should be applied:

   - If an activity mark is ≤ 7 then the final mark of the course (OPO) can be maximum 9/20.
   - If an activity mark is ≤ 5, then the final grade of the course (OPO) can be maximum 7/20.

   Return a list with the OPO final marks for the course. Make sure that the marks do not have a decimal value.

   Note that if the OPO final mark is changed (decreased) based on the 5/7/9 rule, this should be indicated by a string in the list with the following format "X<-Y", where X is the altered mark and Y is the original mark before the rule was applied.