Python Program Documentation – Swimming Calculator

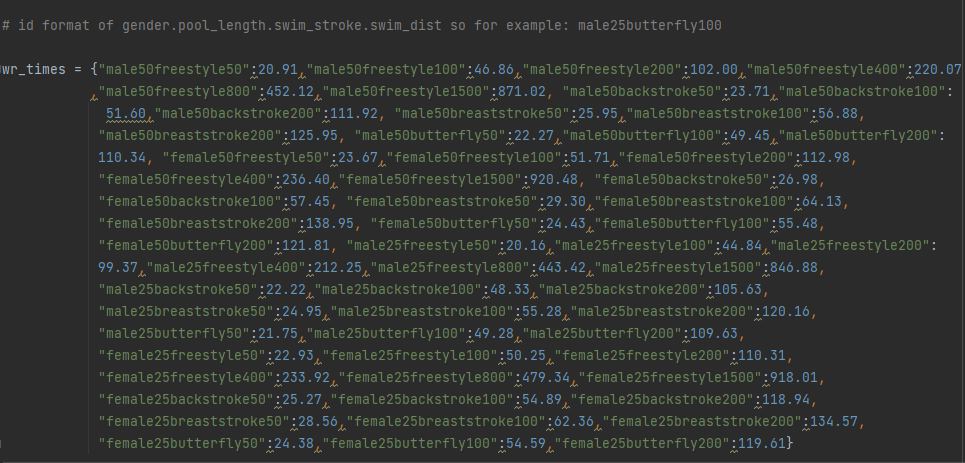
Link to Trello Board: <https://trello.com/invite/b/6Syjnm27/2be8c53fe9c2929a93f88a80f5763915/python-program>

Link to GitHub repository:

<https://github.com/samkley/Swimming_Calculator_OTC_Digital_Science>

**Programming phase:**

I started writing up the dictionary which holds all world record swimming times found in Wikipedia. 

I did not format the dictionary whilst writing it, I did this afterwards to save time. The key in the dictionary is a unique identifier (more on this later) and the value the world record time for that specific identifier.

After finishing the dictionary, I started coding all easygui interfaces such as the various enterboxes or choiceboxes and the identifier calculation. I also coded the if/else statement to check whether the users want to start the program or rather exit it. If they wish to exit it, they were asked to confirm their choice. Whilst coding the easygui interfaces I made sure to keep my implication of health and safety in mind. The health and safety implication were very important to me as I wanted to ensure that absolutely no health issues can be caused by my program. I achieved this by using different types of enter boxes. This meant instead of having to do repeated actions like typing on the keyboard the user is greeted with different types of “tasks”. They must click with the mouse, select an option or type a number. This meant they won’t have to do the same thing (using the keyboard for every input or mouse) and will rather have a variety of tasks to complete. This avoids repetition and muscle strain, discomfort or even injuries. At the same time the variety of easygui boxes also reduce the likelihood of user annoyance as they will not have to do the “same” thing repeatedly.Text

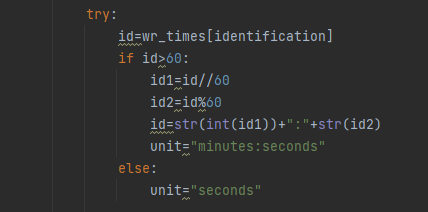
Description automatically generated

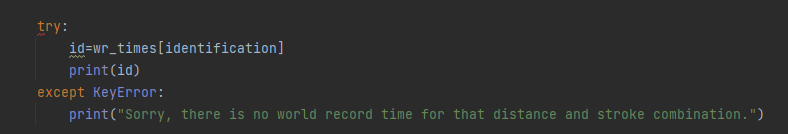
Text

Description automatically generated

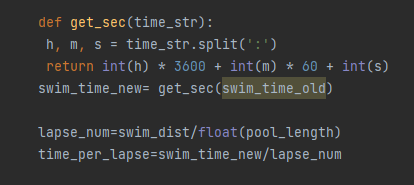
The world record time associated to the user’s input had to be found in the dictionary and its value had to be outputted to the user. The users’ inputs/choices (such as gender, stroke, …) were added together to create an identification string. This string is unique to each input and relates to exactly one world record and time. Using this string the world record time for the user’s input could easily be found as it acted like the key with the time being the value. I did this as following.

To avoid crashes of the program due to identifiers with no world record time, I included the try/except statements to catch a possible error. In this case a KeyError is the only one. If the id of the user does not match an id inside the dictionary the program will print that there is no world record for that specific time. However, this later had to be adjusted so that this error would be output in an easygui box as users would not have been able to read it otherwise. The world record times were saved as seconds and decimals of seconds in the dictionary. I wanted for the world record times to be displayed as MM:SS.SS. To achieve this, I calculated the minutes and seconds+decimal seconds separately from each other. In the end both these values were added together. But this obviously only had to be done if the world record time is larger than 60 seconds (one minute) which meant that I had to add if/else statements that checked whether this was the case. According to if it is bigger than a minute or not, the unit of it had to be changed which will be visible in the final output.



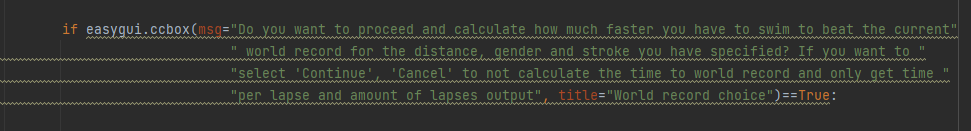


I then started to code all calculations which would calculate the things the users initially wanted to calculate.



Because the originally entered swim time is in the format of HH:MM:SS it had to be converted to purely seconds (get\_sec function) to be useful for further calculations.

The user then had to be asked whether they want to continue with the world record time calculation or just wish to get the above variables output.



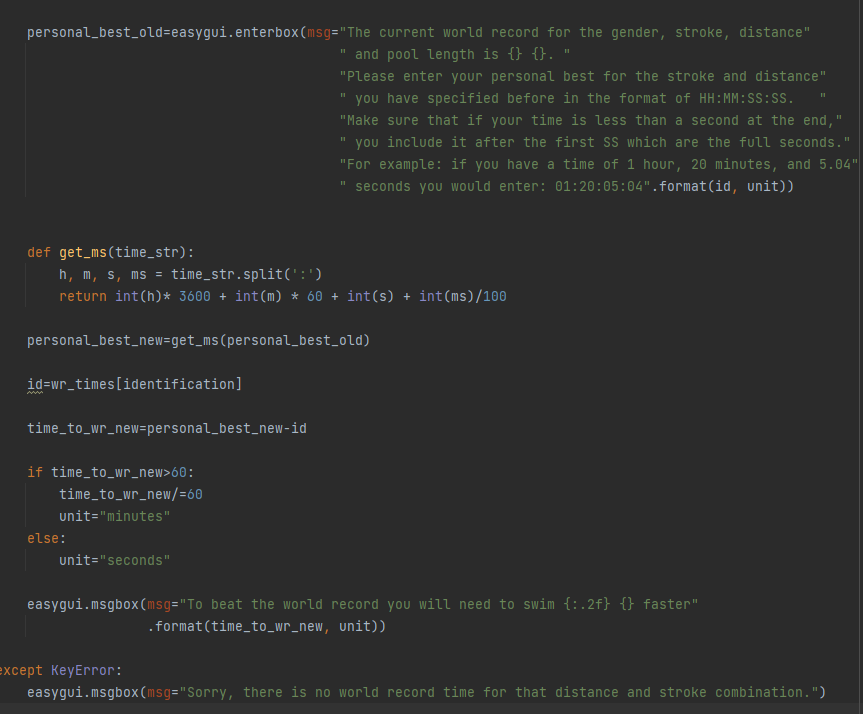
Text

Description automatically generated

If they wish to get their required time to the world record the world record time for the user’s entered data was found first.

Then the user had to be asked what their personal best is for the previously specified stroke, distance, pool length.Text

Description automatically generated

As this was entered in the format of HH:MM:SS.SS for ease of use during input, it later had to be converted to seconds for calculations. Otherwise, it could not be used for the subtraction. I used the same function as the get\_sec function but added the decimal seconds to the function. I then wrote the calculation for the time to the world record and changed the unit of the difference according to if it was bigger than a minute or less than one. The final time to the world record was then output.

Finally, the previously calculated variables such as number of lapses was output in one ContinueCancel box. I chose a ccbox as it allows the user to re-run the program to calculate using a different set of data of for example a friend.

Text

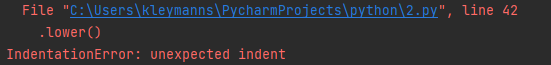
Description automatically generated

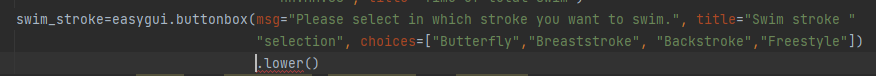
However, if they do not wish to do this the program will be quit.

At the very end the whole code had to be run which I achieved by putting the whole code into a function and calling it.



**Testing, trialling and improving phase:**

After finishing coding, the program and some last adjustment, I tried to run the code but was given an error. 



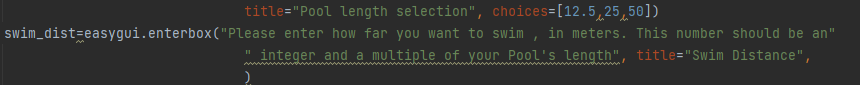
I fixed the error by relocating the indent.



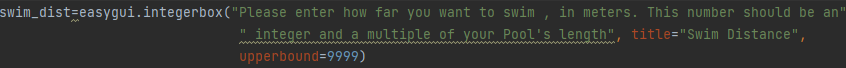
Now running the code does not return any errors with normal (expected) values.

I then tested the program with more expected, boundary and invalid values. These values could only be changed in boxes where the user must enter something and not just choose.

The first issue I came across is that users could enter invalid or insensible values for the swimming length box.



This is because an enterbox does not limit the user in any way. I changed the box to be an integerbox with an upper bound limit.



I later added the lower bound of 25 as anything below 25 metres is very unlikely to be swam for time taking, calculating any values for this would be insensible.

A user can now enter a distance only smaller than 10 kilometres (anything above that is highly unlikely to be swam in one session, for reference Ironman is 3.8 km). Otherwise, they will be met with an error message asking them to re-enter the value.A picture containing text

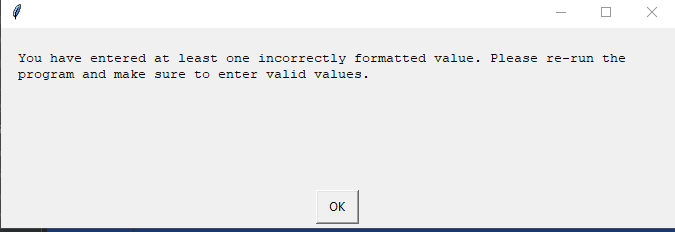
Description automatically generated

Entering invalid values for the swim time enterbox would yield errors further in the calculations in the program (value errors).

I fixed the occurrence of these errors by again adding try/except.

Text

Description automatically generated



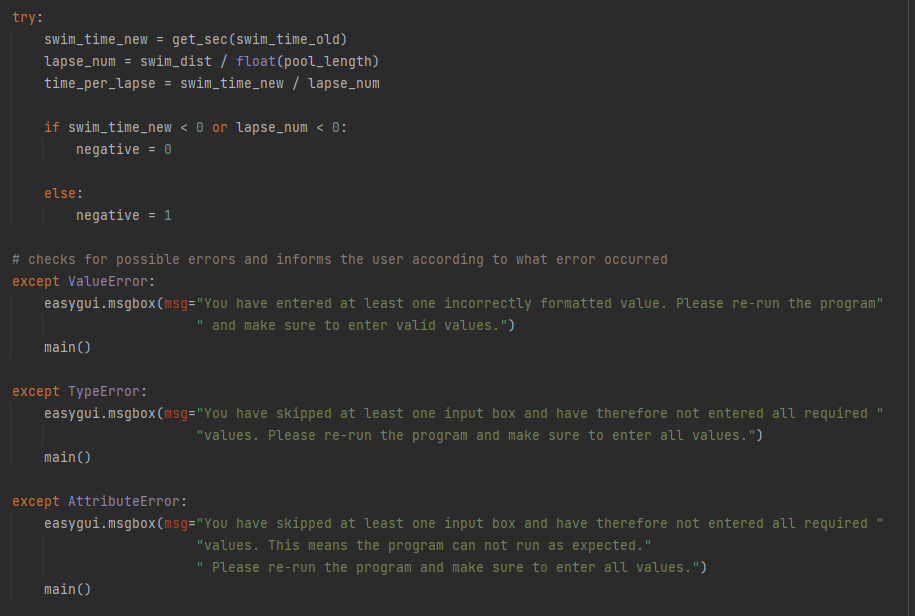
Now if the user entered incorrect values, they would be asked to re-run the program and enter valid values. This ensures that crashes further in the program are avoided.

I have then further tested my program by not entering any values at all which also resulted in crashes. To fix this I expanded my try/except statement.

Now the code will additionally check for type errors (inappropriate data such as no data at all). This will mean that if a user skips an enterbox and fails to enter a required value, the program will catch that error and prompt the user to re-run the program and make sure to enter all values in all boxes. At the same time, I added yet another except statement to check for attribute errors (raised when a function or operation is applied to an object of an incorrect type). The code then looked like this:Text, letter

Description automatically generated

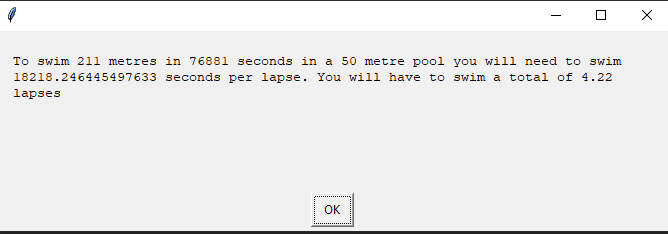
(🡨V.1)



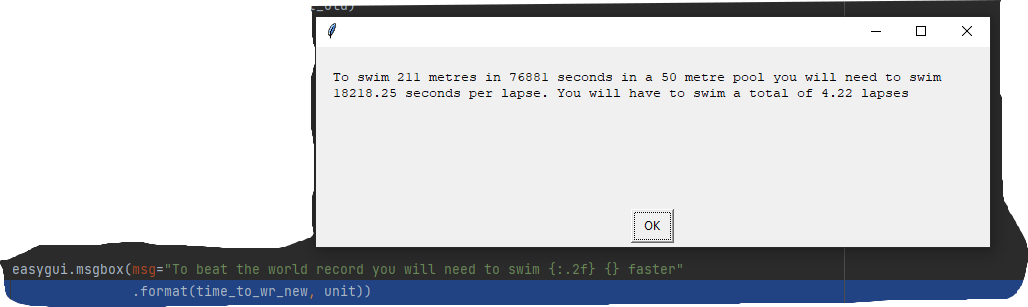
Text

Description automatically generated

This code checks for three different possible errors and avoids program crashes or invalid outputs. One last adjustment I made is to change the quit() to a main() as this will make the program re-run automatically if an error was found instead of having the user find and open the program again. This was done with the functionality implication of my outcome in mind as the code now ensures that the program continues to work without the user having to do unnecessary work. If the program, simply quit itself the user might have a hard time re-opening it which will make the program unfunctional. I then also realised that I should move the get\_sec function to the beginning of the program as this is what the conventions say. I also added an if/else statement which I will explain later.

When entering other expected values (such as 211, 333 or 575), I have come across the error of properly formatting outputs. The output was unrounded and had too many decimal places as the value was taken straight from the division calculation.

A value for seconds does not contain more than two decimal places as any more are useless. I therefore formatted the output to exactly 2 decimal places as following:



Another formatting I have done was for the world record times.

The program now outputs the minutes and seconds for the user specific world record time instead of only outputting seconds/minutes. I have done this as it is more sensible to make the output clearer and easier to understand.A picture containing text

Description automatically generated

Graphical user interface, text

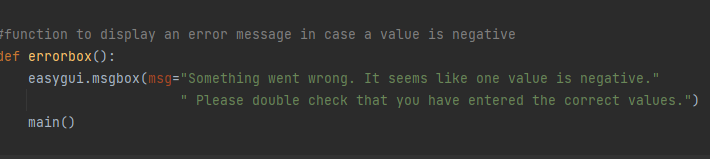
Description automatically generated

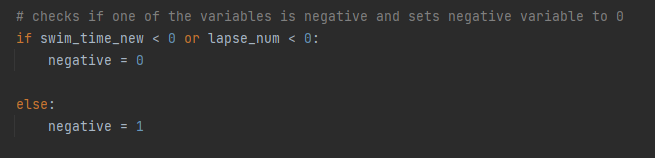
I have also chosen to format outputs to ensure the functionality of my program, an important implication I have discussed before. Numbers with too many digits will become hard to read and memorise which can lead to a decrease in user satisfaction and less functionality as numbers will be read/memorised wrong. Formatting to two decimal places ensures accuracy (more digits not necessary either way) and functionality. Another way I have ensured functionality is by eliminating the possibility of user error by using as previously discussed specific boxes that do not allow invalid values to be entered or by using try/except to catch errors and prompt the user to fix them.Graphical user interface, text, application

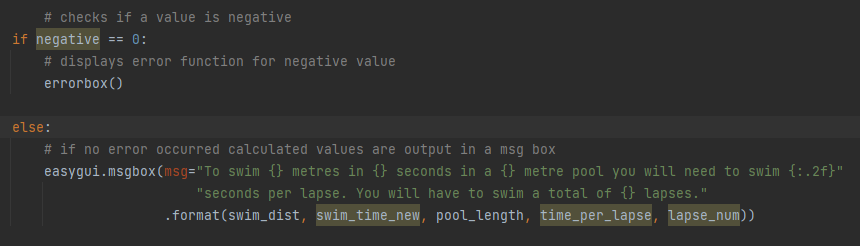
Description automatically generated

(Example of how invalid numbers, -1, do not work due to the box being an integerbox with a lower bound of 25 🡪 25 as anything less is not being swam, at least two lapses of a 12.5 metre pool, anything less is not used for times)

At the very end I have updated the program to also spot negative values of the swim time or the time per lapse. Before adding this my program would output insensible values such as a time per lapse of -10 seconds. If these values are negative, it can only be because the user entered a negative value beforehand. So before negative and wrong values are displayed the program catches the error and prompts the user to re-enter the value. As the negative error could occur in two different occasions, I wrote a function to display the box with the error message when necessary.







(Checking if negative variable is set to 0 as if it is a variable must be negative and an error message must be shown)

I named a variable negative which will become 0 if one value that was calculated is negative. If a value that was calculated is negative it means that the user entered at least one negative value. I chose to create a variable for this as if I ever update my program to include more calculations, I can simply add the new variable to the if … < 0 statement and if it is negative the program will count it as an error. I will not have to go through the code and add the new variable to each statement checking whether it is less than 0 which I would have to do if I didn’t use a variable but rather checked if variable < 0 every time. This decision makes my program more robust.