DEVELOPING THE CODED SOLUTION

For my development of the coed solution I have decided to split the process into separate milestones. Each milestone will be a significant part of the code that once completed I will move on from onto the next milestone.

Milestone 1 – Creating the basic GUI and basic customisation

Milestone 2 – Creating CPU tests

Milestone 3 – Creating GPU tests

Milestone 4 – Showing pc information

Milestone 5 – Adding in my Component database

Milestone 6 – Improving Visual appearance of GUI

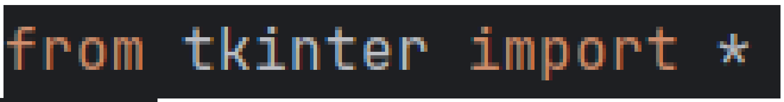
Milestone 7 – Improving Running of the program

MILESTONE 1

SETTING UP A WINDOW AND INITIAL RESEARCH

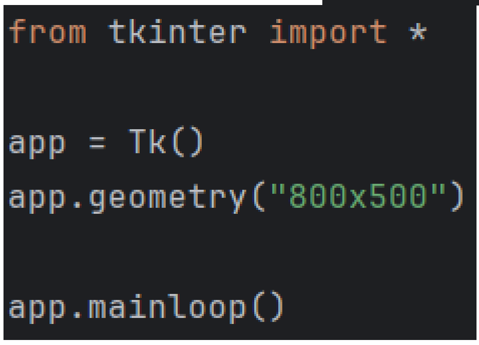
I started my development process by accessing my analysis objectives the first of which was creating a visually pleasing Gui. I first decide on using python tkinter as it is the most well known Gui development suite and can be used for both simple programs and complicated ones. I found out about import libraries and used this feature in my program.

I researched some basic customisation so that I could fulfil the requirements of having a good visually pleasing Gui which was not only an initial aim of mine but was also another request by my stakeholders and



```
from tkinter import *
```

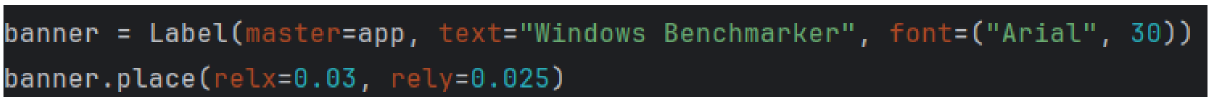
those



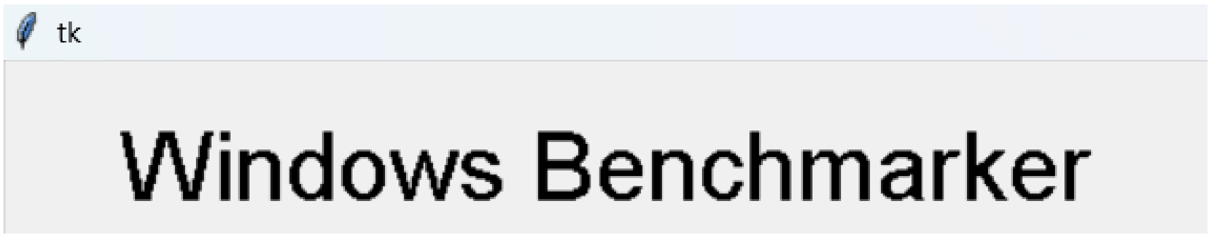
```
from tkinter import *

app = Tk()
app.geometry("800x500")

app.mainloop()
```

who I interviewed. I changed some colours and renamed the window. After this I discovered that I could resize the window as I pleased and consequently changed the sizing so that the user didn't need to enlarge the window improving the experience.

Then I added a title using the Label function in tkinter positioning it using padx and pady.



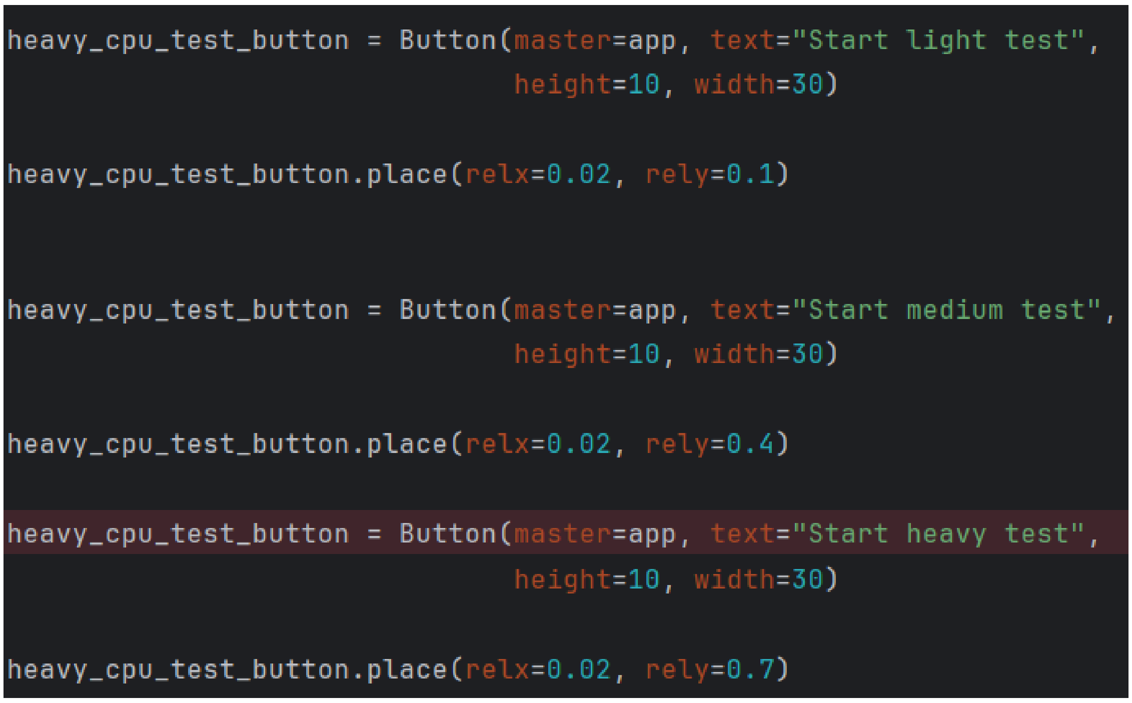
```
banner = Label(master=app, text="Windows Benchmarker", font=("Arial", 30))
banner.place(relx=0.03, rely=0.025)
```

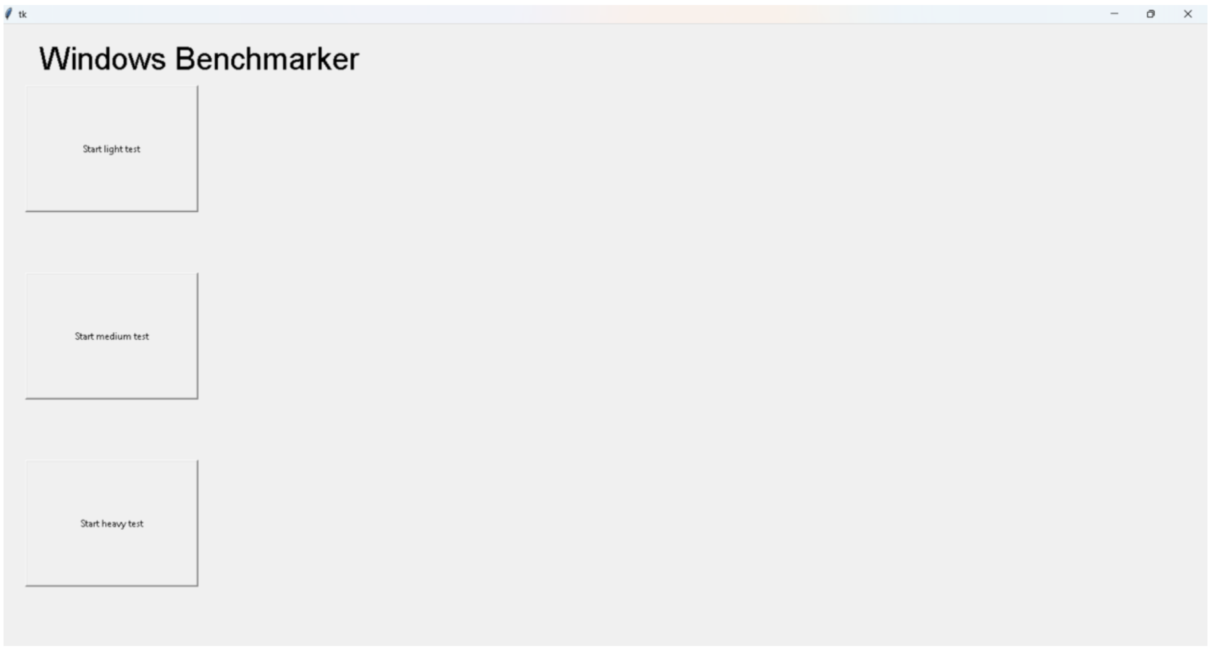


MILESTONE 2

COMPUTER SCANS

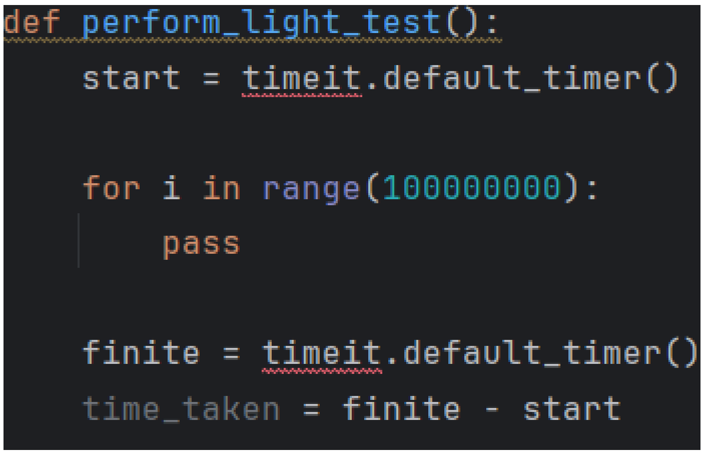
Another one of my requirements as stated in the analysis section was to have a basic computer scan as well as thorough scan and an intense scan. For The user to actually be able to use these buttons where needed. So, I set about creating a button using the button function within tkinter. I then resized the button to fit properly in the window and colouring it so that it stood out. I then added 2 more buttons for the other levels of tests.


```
heavy_cpu_test_button = Button(master=app, text="Start light test",
                               height=10, width=30)

heavy_cpu_test_button.place(relx=0.02, rely=0.1)


heavy_cpu_test_button = Button(master=app, text="Start medium test",
                               height=10, width=30)

heavy_cpu_test_button.place(relx=0.02, rely=0.4)

heavy_cpu_test_button = Button(master=app, text="Start heavy test",
                               height=10, width=30)

heavy_cpu_test_button.place(relx=0.02, rely=0.7)
```



HOW I TEST CPU

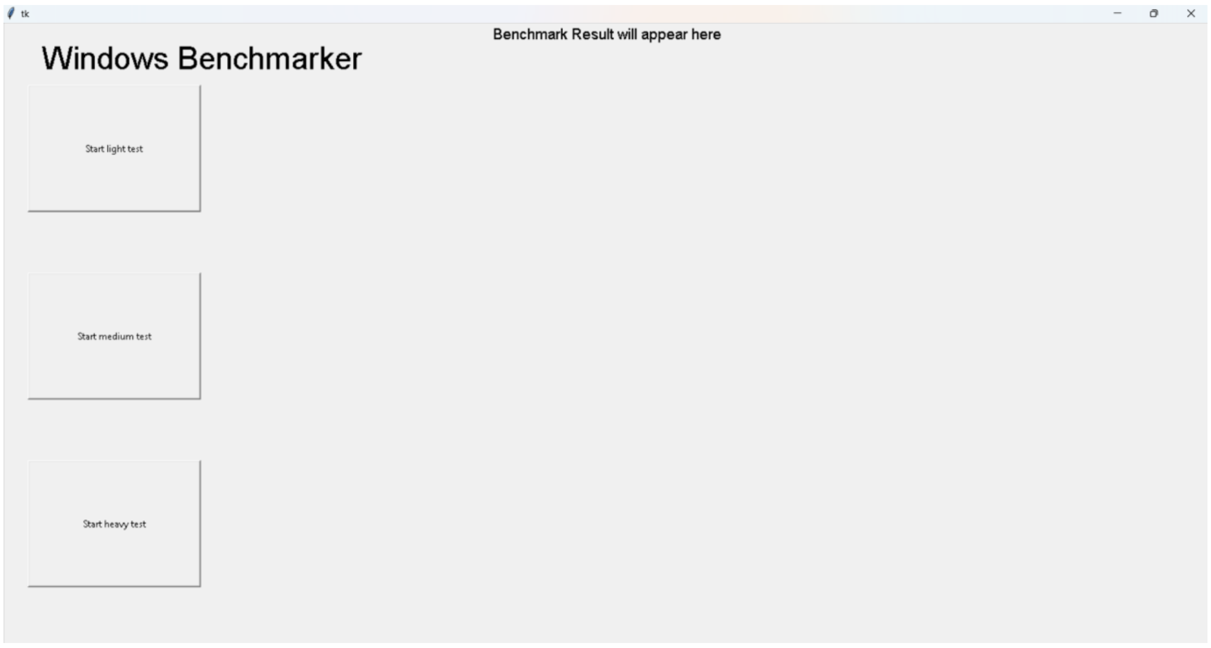
Since these currently didn't do anything, I started coding on the actual function to run a test on the CPU. From some research I found that the best method for me to test the CPU would be to run a series of mathematical operations. I made my program count to a certain number for each different test with the most intense test needing to count to the highest number.



```
def perform_light_test():
    start = timeit.default_timer()

    for i in range(100000000):
        pass

    finite = timeit.default_timer()
    time_taken = finite - start
```

DISPLAYING CPU INFORMATION TO USER

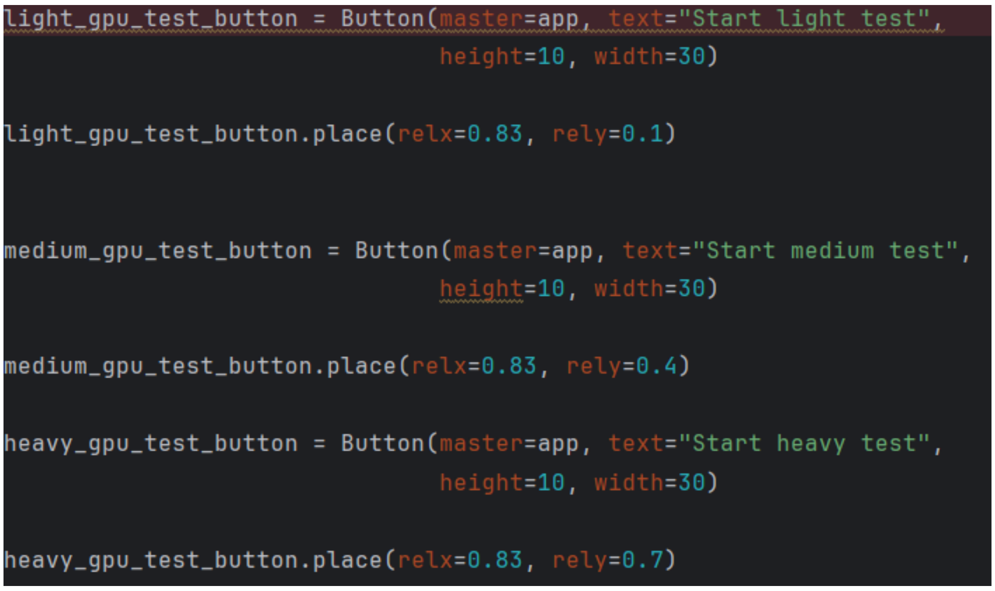
At this stage my program could benchmark the CPU however it was unable to show this information to the user. So, using the label function again I created an execution time display. Linking this to timetaken (the variable used to store how long it took to benchmark the CPU) I was able to display it.



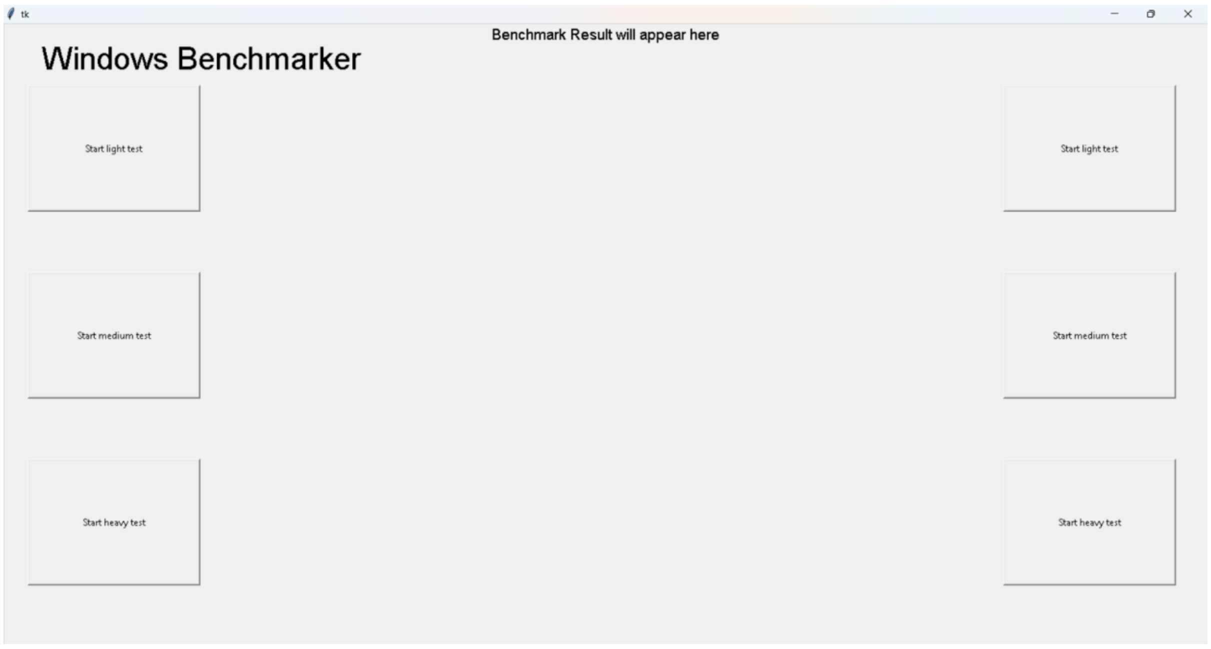
MILESTONE 3

GPU TEST BUTTONS

Another important part I knew I needed to include was a Gpu test. So, replicating the same buttons from the CPU test I added Gpu test buttons on the right-hand side of the GUI.

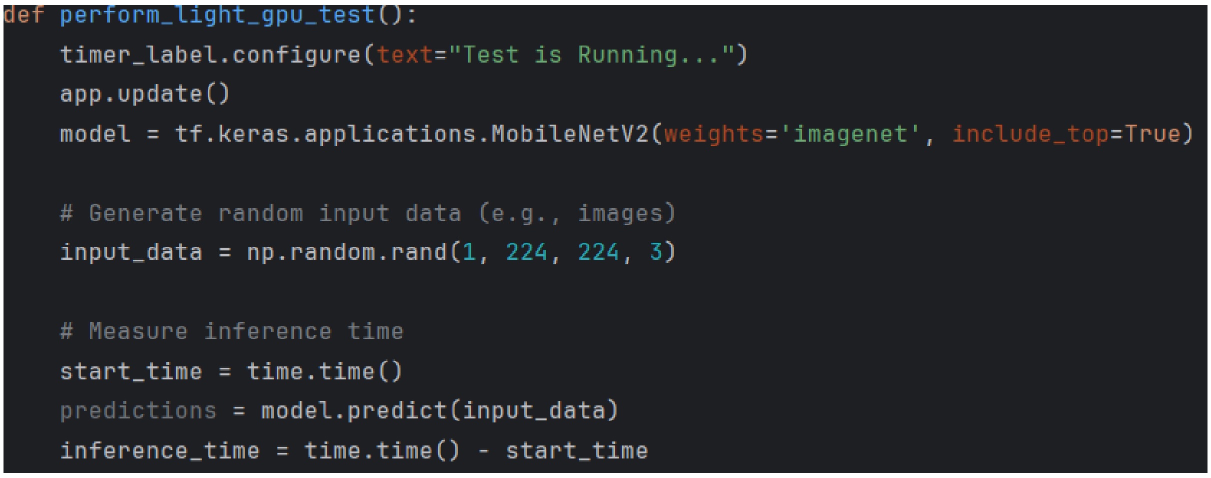


```
light_gpu_test_button = Button(master=app, text="Start light test",
                               height=10, width=30)

light_gpu_test_button.place(relx=0.83, rely=0.1)


medium_gpu_test_button = Button(master=app, text="Start medium test",
                                height=10, width=30)

medium_gpu_test_button.place(relx=0.83, rely=0.4)

heavy_gpu_test_button = Button(master=app, text="Start heavy test",
                               height=10, width=30)

heavy_gpu_test_button.place(relx=0.83, rely=0.7)
```

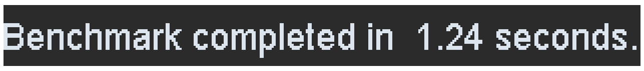


MAKING GPU BUTTONS WORK

After this I added functionality to these buttons using the keras library, which is used for data automation, model tracking and crucially for me, performance monitoring.



```
def perform_light_gpu_test():
    timer_label.configure(text="Test is Running...")
    app.update()
    model = tf.keras.applications.MobileNetV2(weights='imagenet', include_top=True)

    # Generate random input data (e.g., images)
    input_data = np.random.rand(1, 224, 224, 3)

    # Measure inference time
    start_time = time.time()
    predictions = model.predict(input_data)
    inference_time = time.time() - start_time
```

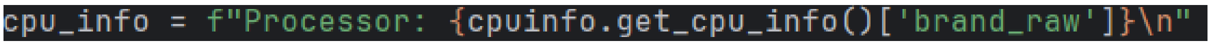


```
Benchmark completed in  1.24 seconds.
```

MILESTONE 4

SHOWING USER INFORMATION USING CPU INFO

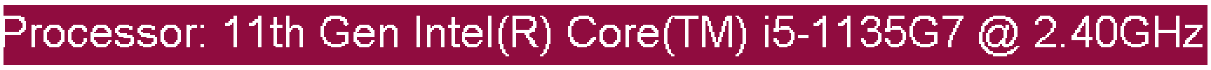
Another main requirement that I knew I needed to include was to show the user their system information (ram, CPU, gpu) The first one I implemented was the CPU information. I did this by using the CPU info library which detects essential for performance optimization information about host CPU.



```
cpu_info = f"Processor: {cpuinfo.get_cpu_info()['brand_raw']}\n"
```

DISPLAYING CPU INFORMATION TO THE USER

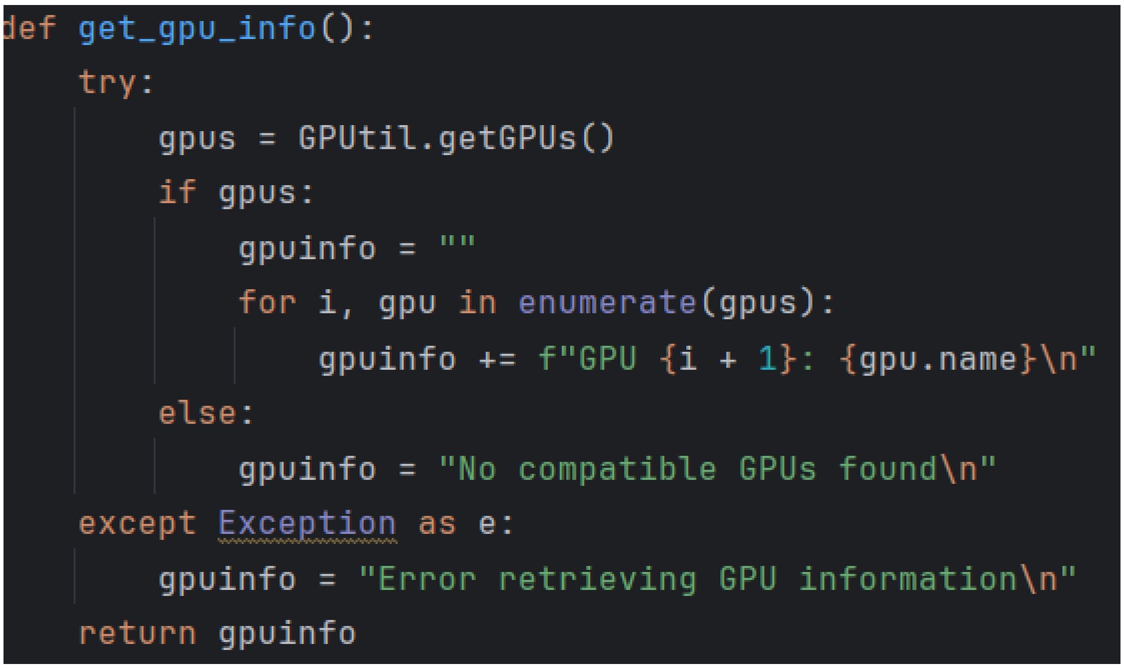
I then used the Label function like previous to display this in my Gui, again using padx and pady to position it accordingly.



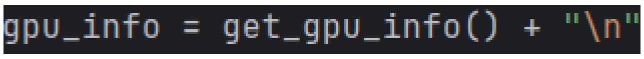
```
Processor: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
```

DISPLAYING GPU INFORMATION TO THE USER

After Completing the CPU information, I decided to show the gpu information. Although I knew I had to use the label function to display this information I was unsure on how to find the gpu information in computer. Luckily, I found a library called GPUtil which locates all GPUs on the computer, determines their availability and returns an ordered list of available GPUs.



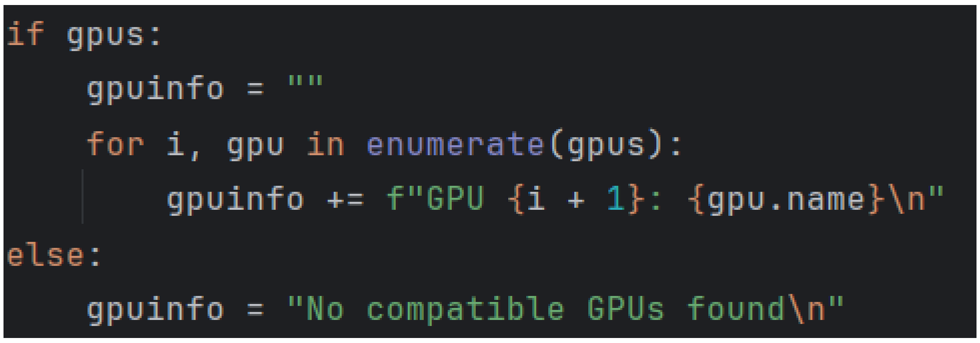
```python
def get_gpu_info():
    try:
        gpus = GPUtil.getGPUs()
        if gpus:
            gpuinfo = ""
            for i, gpu in enumerate(gpus):
                gpuinfo += f"GPU {i + 1}: {gpu.name}\n"
        else:
            gpuinfo = "No compatible GPUs found\n"
    except Exception as e:
        gpuinfo = "Error retrieving GPU information\n"
    return gpuinfo
```



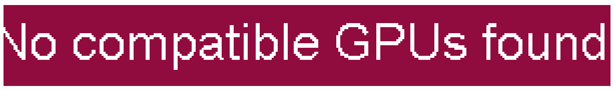
```python
gpu_info = get_gpu_info() + "\n"
```

DETECTING/NOT DETECTING GPUS

One problem I soon encountered was with computers that don't have a dedicated gpu. My program would get confused and give an error. I quickly changed this to tell the user that they didn't have a gpu if none was detected.



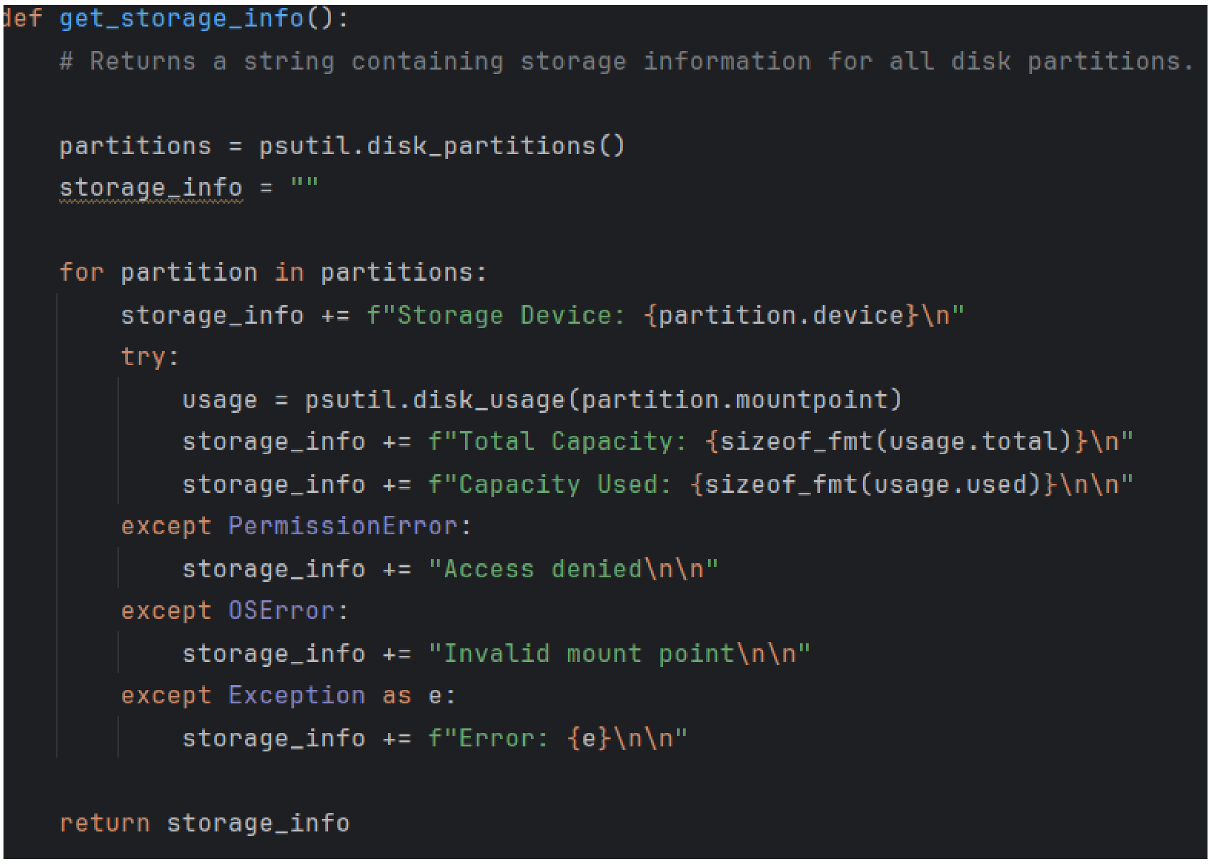
```python
if gpus:
    gpuinfo = ""
    for i, gpu in enumerate(gpus):
        gpuinfo += f"GPU {i + 1}: {gpu.name}\n"
else:
    gpuinfo = "No compatible GPUs found\n"
```



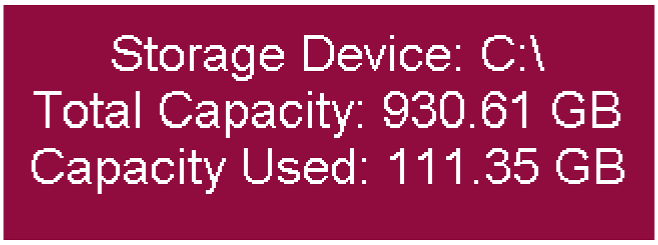
```
No compatible GPUs found
```

DISPLAYING STORAGE INFORMATION TO THE USER

Next up was the storage information. I wanted to show the user both the total amount of storage their computer had and also the amount of storage they had used up as this is another common feature in another benchmarking software. Luckily for me I found out about Psutil in the same forum as GPUtil. Psutil allows for retrieving of information on running processes and system utilization (CPU, memory, disks, network, sensors) and by locating the disk partitions on the user's system you are able to identify how large the disk is and how much of it is filled up which was exactly what I needed.



```
def get_storage_info():
    # Returns a string containing storage information for all disk partitions.

    partitions = psutil.disk_partitions()
    storage_info = ""

    for partition in partitions:
        storage_info += f"Storage Device: {partition.device}\n"
        try:
            usage = psutil.disk_usage(partition.mountpoint)
            storage_info += f"Total Capacity: {sizeof_fmt(usage.total)}\n"
            storage_info += f"Capacity Used: {sizeof_fmt(usage.used)}\n\n"
        except PermissionError:
            storage_info += "Access denied\n\n"
        except OSError:
            storage_info += "Invalid mount point\n\n"
        except Exception as e:
            storage_info += f"Error: {e}\n\n"

    return storage_info
```



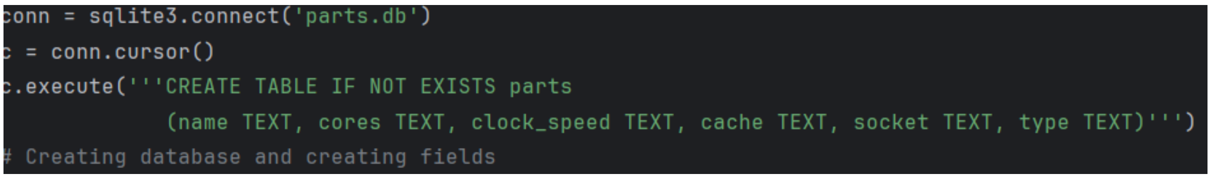
```
Storage Device: C:\
Total Capacity: 930.61 GB
Capacity Used: 111.35 GB
```

MILESTONE 5

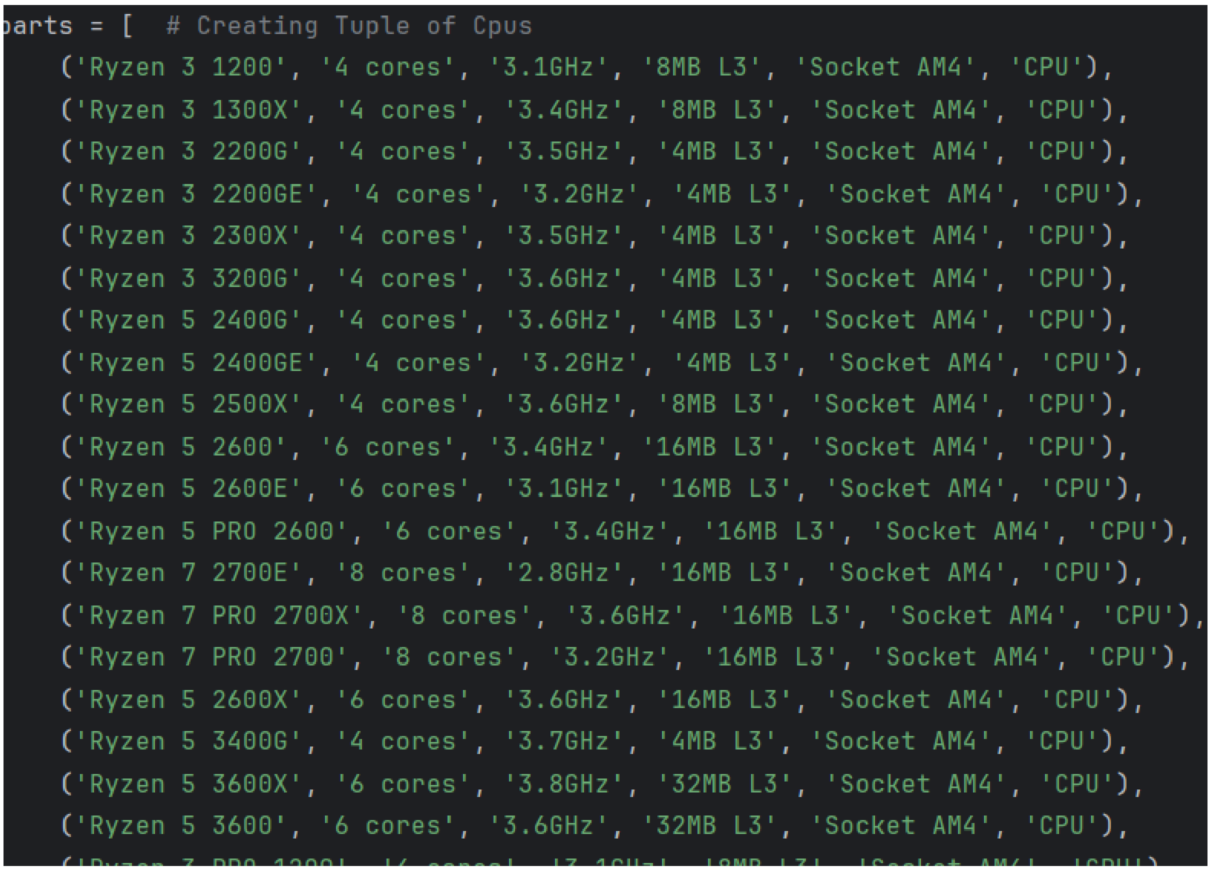
PC COMPONENT DATABASE

My last main feature was my database on pc components. I decided to only include CPUs and GPUS as these are the main components that my target audience would be interested in. Upon research online I also found out that many Pc experts regard all other components as being not as different from each other. Using my prior knowledge in SQLite I was able to create a database with a primary key and multiple fields. SQLite can be used to create a database, define tables, insert and change rows, run queries and manage a database file. It also serves as an example for writing applications. It is commonly used in python programs and is widely regarded as one of the best database building libraries. Once I had named my database and added my fields, I started to fill out my database with components. I used

PCpartpicker, A well-known website that shows a comprehensive comparison of thousands of parts to ensure my database included every possible component that my users could want to look up.



```
onn = sqlite3.connect('parts.db')
 = conn.cursor()
.execute('''CREATE TABLE IF NOT EXISTS parts
             (name TEXT, cores TEXT, clock_speed TEXT, cache TEXT, socket TEXT, type TEXT)''')
 Creating database and creating fields
```

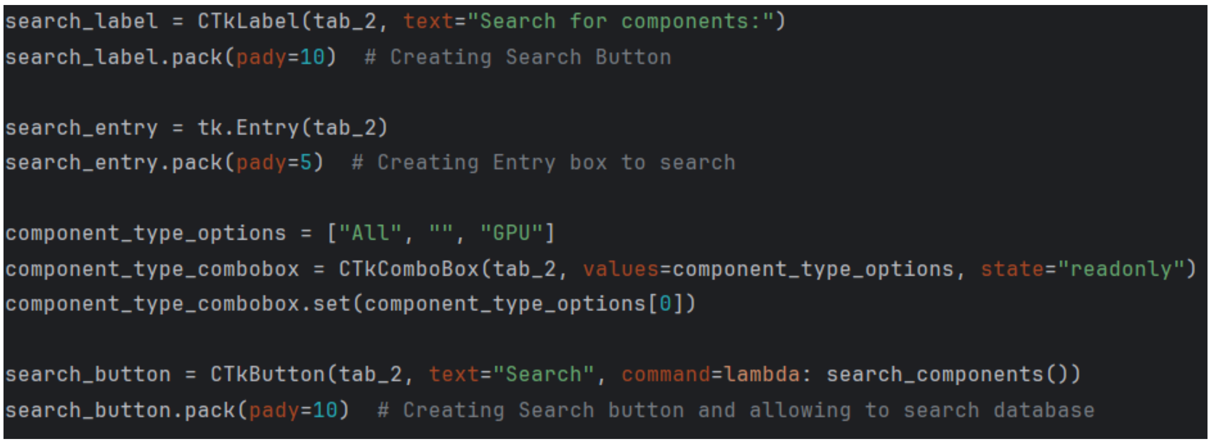


```
arts = [  # Creating Tuple of Cpus
    ('Ryzen 3 1200', '4 cores', '3.1GHz', '8MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 3 1300X', '4 cores', '3.4GHz', '8MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 3 2200G', '4 cores', '3.5GHz', '4MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 3 2200GE', '4 cores', '3.2GHz', '4MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 3 2300X', '4 cores', '3.5GHz', '4MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 3 3200G', '4 cores', '3.6GHz', '4MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 5 2400G', '4 cores', '3.6GHz', '4MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 5 2400GE', '4 cores', '3.2GHz', '4MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 5 2500X', '4 cores', '3.6GHz', '8MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 5 2600', '6 cores', '3.4GHz', '16MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 5 2600E', '6 cores', '3.1GHz', '16MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 5 PRO 2600', '6 cores', '3.4GHz', '16MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 7 2700E', '8 cores', '2.8GHz', '16MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 7 PRO 2700X', '8 cores', '3.6GHz', '16MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 7 PRO 2700', '8 cores', '3.2GHz', '16MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 5 2600X', '6 cores', '3.6GHz', '16MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 5 3400G', '4 cores', '3.7GHz', '4MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 5 3600X', '6 cores', '3.8GHz', '32MB L3', 'Socket AM4', 'CPU'),
    ('Ryzen 5 3600', '6 cores', '3.6GHz', '32MB L3', 'Socket AM4', 'CPU'),
```

(More cpus are included however the list is too long to include here)

SEARCH FEATURE IN MY DATABASE

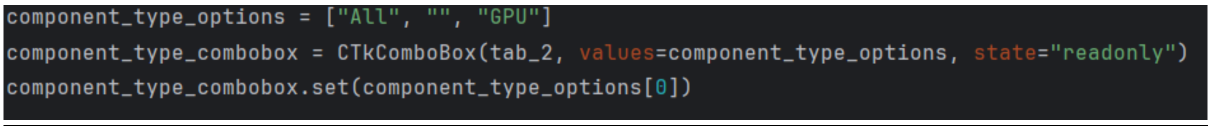
I then started work on the search feature that allows my user to access the database and search it to find the components they are looking for. I first defined a function `search_components()` that handles searching for components in a database based on user input. It executes SQL queries to fetch relevant results from the database, clears the existing display, and populates a treeview with unique search results to show to the user, avoiding duplicates. The function ensures that the search results are based on the user's query and the selected component type, providing a streamlined way to display component information in the app.



```python
search_label = CTkLabel(tab_2, text="Search for components:")
search_label.pack(pady=10)  # Creating Search Button

search_entry = tk.Entry(tab_2)
search_entry.pack(pady=5)  # Creating Entry box to search

component_type_options = ["All", "", "GPU"]
component_type_combobox = CTkComboBox(tab_2, values=component_type_options, state="readonly")
component_type_combobox.set(component_type_options[0])

search_button = CTkButton(tab_2, text="Search", command=lambda: search_components())
search_button.pack(pady=10)  # Creating Search button and allowing to search database
```

ADDING SEARCH FILTER TO DATABASE

I also added a filter so that the user could filter between "all","CPUs" or "GPUS". This was done by assigning each entry in the database a "type" referring to what type of component they were. I.e. CPU/CPU. When CPU or gpu were selected as a filter the database checks each entry in the database to see if they fall in that "type" and then either displays them or not.



```python
component_type_options = ["All", "", "GPU"]
component_type_combobox = CTkComboBox(tab_2, values=component_type_options, state="readonly")
component_type_combobox.set(component_type_options[0])
```

INTERVIEW PLAN

As more project progressed I decided to have another interview to see my users their opinions on my project as well as an updated idea on what was necessary to add to it and what was no longer needed.

Interview

Interviewees: Abbas Davdani and Ali Amiri

Question – What are your general opinions on my current gui?

Answer: The layout of the GUI has good placement of elements, however, the buttons are sharp on the eyes and the lack of a colour theme makes the GUI look slightly less user-friendly.

Answer2: Answer - The way you designed the gui is very good, the layout looks easy to navigate and use. The CPU test is both effective and accurate, but it seems there's a hiccup with the GPU test not functioning. As for improvements, refining the buttons to give them a more polished and professional appearance would elevate the overall aesthetic. Additionally, enriching the database by including detailed specifications of each component would empower users to make more informed decisions regarding which components best suit their needs.

Overall, the GUI is progressing admirably, and with these tweaks, it's poised to deliver an even more exceptional user experience.

Question – Do you feel any important features are parts missing?

Answer: The benchmarker is good and shows good potential, however, it is missing some key elements which are present in other modern benchmarkers, including a GPU test.

Question – In its current state is this program one you would use?

The benchmarker program will be useful for determining the performance of my system. The application shows a lot of potential and it is quite likely that I will use it in the future.

Question – When completed do you think this program will be to the level of other benchmarkers?

Answer: The current status of the application is quite good and I believe that it has the potential to become as good as any other benchmarker on the market.

Using the information and suggestions provided by my interviewees I decided to amend a few things. I decided to fix the GPU test and improve the look and feel of my program by adding colour, improving my buttons and addingtabs for easy navigation. I was pleased to see that my users saw that my program had good potential and would have the possibility to be as good as many popular benchmark tools. In terms of the databas I decided to to add more sections and more specifications so the users can compare with other component and find something that suited their needs.

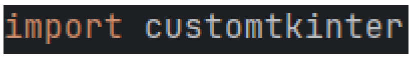
MILESTONE 6

USING CUSTOMTKINTER TO CUSTOMIZE MY GUI

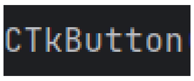
Now that my essential features were all complete my attention turned to making my program more aesthetically pleasing and easier to navigate. While looking for ways to customize a tkinter GUI I discovered a YouTube video that discussed using CustomTkinter which is a tkinter extension which provides extra UI-elements like the CTkButton, which can be used like a normal tkinter button, but can be customized with a border and round edges.

CHANGING MY WINDOW FROM TKINTER TO CUSTOMTKINTER

The first change I had to make was importing customtkinter and setting my window as a customtkinter window rather than tkinter. Then I had to change all my buttons from tkinter buttons to cutomtkinter buttons using CTkLabel which is the label function from customtkinter. It allowed me to change the border size font and positioning.



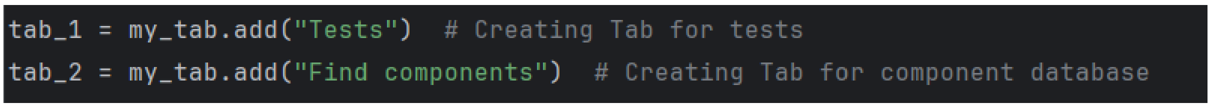
```
import customtkinter
```



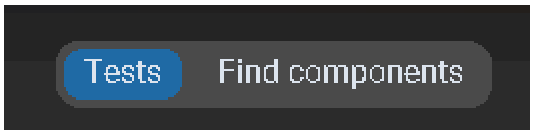
```
CTkButton
```

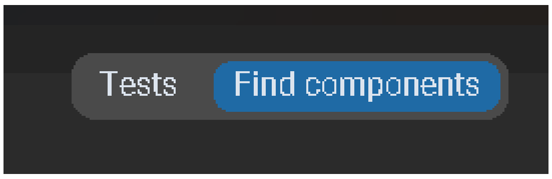
ADDING TABS

To make more program less cluttered and more organised I decided to add multiple tabs to my program. Since the most essential features were the buttons to run the tests and the computer information, I decided to have them in the first tab and move the database to the second tab. I did this by using the tab feature in Customtkinter that allows you to add multiple tabs and customize their names, how they look and where they are positioned.



```
tab_1 = my_tab.add("Tests")  # Creating Tab for tests
tab_2 = my_tab.add("Find components")  # Creating Tab for component database
```



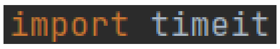


MILESTONE 7

<u>Timing How long it takes for tests to run</u>

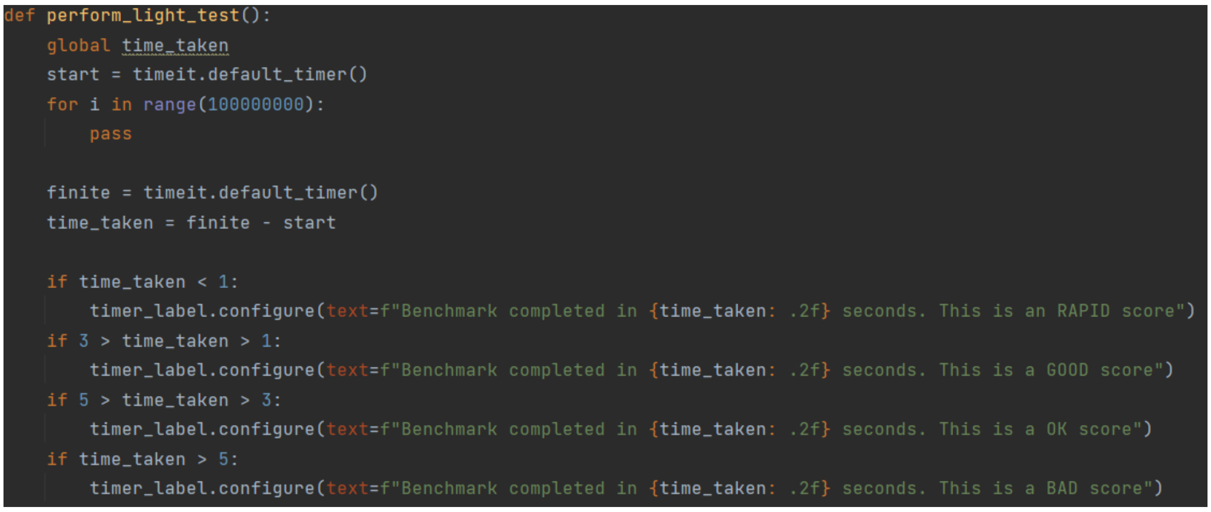
I previously stated that I wanted to tell the user how their system was performing in comparison to other systems, so I decided to implement a way to do this. By importing the timeit function I was able to time how long the user's system took to run each test. When the function was run it would start a timer and by using an if statement, I was able to identify which range of time the user's pc took to run the test.

IMPORTING THE TIMEIT LIBRARY



```
import timeit
```

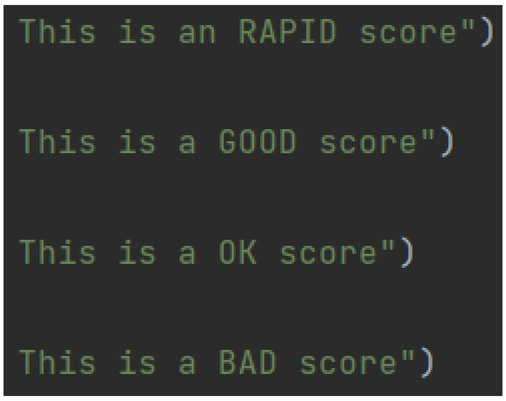
The code to find out what score the user achieves (in a light test. Same method of code generation is used for heavy and medium tests but with adapted values for time taken)



```
def perform_light_test():
    global time_taken
    start = timeit.default_timer()
    for i in range(100000000):
        pass

    finite = timeit.default_timer()
    time_taken = finite - start

    if time_taken < 1:
        timer_label.configure(text=f"Benchmark completed in {time_taken: .2f} seconds. This is an RAPID score")
    if 3 > time_taken > 1:
        timer_label.configure(text=f"Benchmark completed in {time_taken: .2f} seconds. This is a GOOD score")
    if 5 > time_taken > 3:
        timer_label.configure(text=f"Benchmark completed in {time_taken: .2f} seconds. This is a OK score")
    if time_taken > 5:
        timer_label.configure(text=f"Benchmark completed in {time_taken: .2f} seconds. This is a BAD score")
```

CREATING "SCORES" FOR EACH TEST

I had to identify how long a "good" score would be or a "bad" score. I decided to have 4 different categories of results. Good, bad, great and ok. Not having to many different types of results simplifies the program and unlike other benchmarkers which have results that can be confusing to the average user such as "spaceship" or "car", I used terms that can be understood by most people. This expands my target audience and continues my plan of making my program as jargon free and visually pleasing as possible.



```
This is an RAPID score")

This is a GOOD score")

This is a OK score")

This is a BAD score")
```

FIGURING OUT HOW TO ACHIEVE THE BEST TIMINGS FOR EACH TEST

I had to find out what a valid time of running my benchmarks would be and had to find out just how long a "good" score would be. The best way to do this I reasoned, was to run the benchmark on many individuals' systems and to find out what time they could achieve in the tests, I had to do this for each level of test as they all take a different amount of time to run. I asked if I could run the benchmark of all 3 tests on my friends' computers, they all have systems of varying compatibility, so I got a large range of results. The best system and worst system became my upper and lower limits, and the average time became what I would consider an "Ok" score.

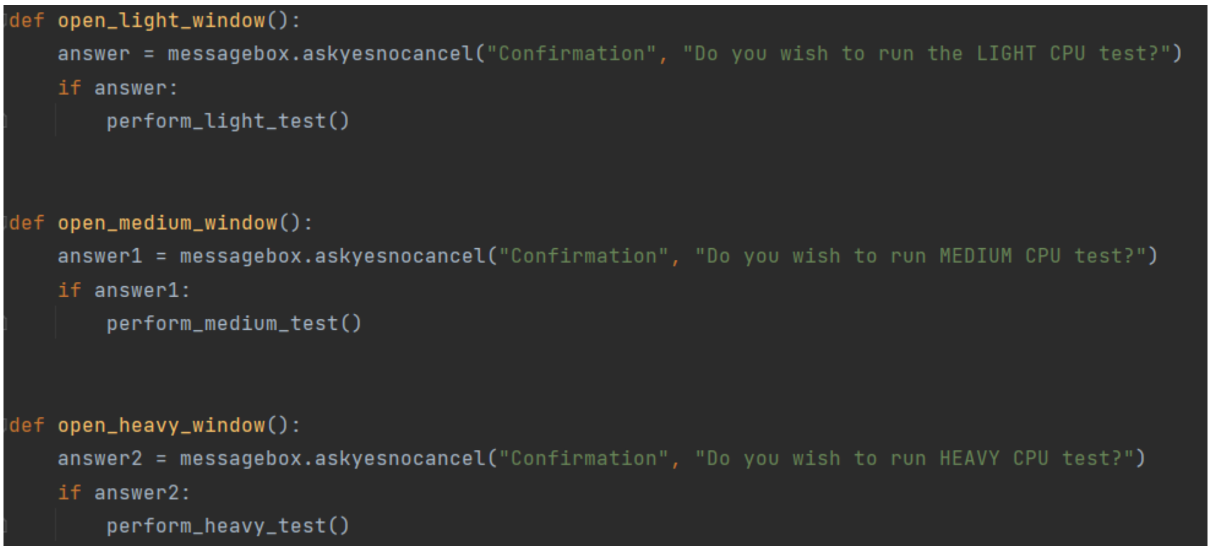
ENSURING MY TESTS ARE RELIABLE FOR EACH SYSTEM.

I did this 3 times on each system for each level of test to make sure I didn't get any anomalous results, but these were far and few between as my benchmarker proved to be a reliable program. The end result was a function that timed the benchmark and compared the value to the values set for each level of test before displaying the time taken and the "score" to the user.

ADDING CONFIRMATION OF RUNNING TESTS

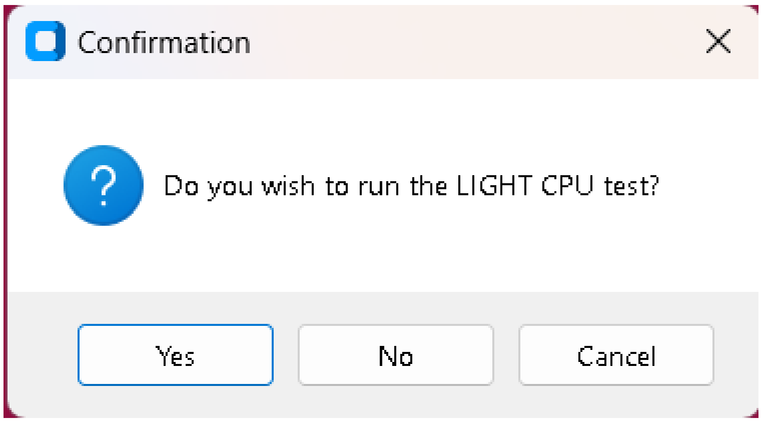
To improve the quality of life of my program and to help users understand what they are doing before they do it, I decided to add a confirmation after pressing the "Run CPU test" and "Run Gpu test". It asks them if they want to run the particular test that they clicked the button of. I decided that it would ask if them if they either wanted to run the test, not run the test, or cancel what they were doing. I used the messagebox module within tkinter and more specifically the yesnocancel element, as it gives the user 3 options, saying yes, no, or cancelling whatever they were trying to do.

Code for opening the window of confirmation:



```
def open_light_window():
    answer = messagebox.askyesnocancel("Confirmation", "Do you wish to run the LIGHT CPU test?")
    if answer:
        perform_light_test()


def open_medium_window():
    answer1 = messagebox.askyesnocancel("Confirmation", "Do you wish to run MEDIUM CPU test?")
    if answer1:
        perform_medium_test()


def open_heavy_window():
    answer2 = messagebox.askyesnocancel("Confirmation", "Do you wish to run HEAVY CPU test?")
    if answer2:
        perform_heavy_test()
```

What the confirmation window looks like:


Do you wish to run the LIGHT CPU test?