GROWING OXYGEN

Nic Harrod A-Level NEA

Candidate Number: 2300

Center Number: 12610

Contents

Analysis

Problem Identification

A common issue many people who own houseplants encounter is deciding what houseplant is best for them. The reason this is such a large issue is that for many plants, if one aspect of how they're cared for changes, the plant could be damaged irreversibly (or killed). Whilst there are many solutions already existing for this, they are either too light-hearted (in the case of online quizzes which are unsuitable for those with high levels of understanding), or they could be behind a paywall in the sense of general plant care apps. However, what I intend to develop is a free piece of software that is accessible too those with both high and low levels of understanding plant care.

Computational methods

The problem lends itself into a computational solution because whilst there are many books on the topic, they still require a lot of patience and don't specifically accommodate to the user's needs. As will be seen below, some of the books found can be very detailed which may scare away those with little understanding. A computational solution can just retrieve a recommended plant from a database instead of indexing through a book. After retrieving the plant from the database due to calculations, a graphical interface can then display it.

Various implementations of computational techniques will be used to solve different problems. Lending myself to these allows me to characterise the problem in a more typical sense.

Problem Recognition

One of the most key pieces of analysis is understanding the actual complex parts of the problem and solution. In this case it would be the calculation of the recommended plant since this is the part that isn't clear. Unlike a graphical or database system which are pretty clear in what they do and how they can be designed since the concept behind them is well established and don't require any new concepts to be created.

Problem Decompostition

This problem can be decomposed into a set of different steps which allows me to take the time to tackle each of them individually and therefore giving me the space to appropriately balance the different aspects of the problem and solution.

1. The database system of plants and their attributes with a series of relational tables.
2. The Graphical User Interface of the actual program that the user interacts with.
3. The calculation which takes inputs from the GUI and outputs data from the database.
4. Integrating the database and calculations so it retrieves queries from the database.
5. Integrating the GUI to the calculations so it can take the format of data from the database and sends the correct data to the calculations.

When these are completed, the program will have been done and all I will need to do is tidy the files since I am probably going to have many unnecessary splitting of code and data.

Divide and Conquer

This application of computational methods and thinking is for when I have split my smaller steps into even smaller functions and parts. The advantage of this is that it makes my solution far more modular and easier to debug and edit this, means if I change one part I won't have to change the entirety of the code. An example of using divide and conquer is by splitting my GUI code into multiple screens so I can edit them individually.

Abstraction

Abstraction is the act of removing unnecessary data to the point that the only data available is vital for solving the problem. This is helpful because the program does not need to know the name of the user or what the plant looks like in order to process calculations, overall increasing the efficiency of the system.

For any given part of the program, only the input and output are needed by any of the components. The GUI doesn't need to know what the database looks like, all it needs is the form of the data sent by the algorithm. Similarly, the database doesn't care about how data is eventually processed, just what shape the queries it gets sent are.

Potential limits of the solution

There are a few limitations of this solution for the problem stated. One of these limits is potential advice for any plants the user already has, although I intend to add care instructions to the plant recommendations. These are most likely going to be quite short and also only have things like watering and potting advice. Whilst these could give help to the user, the thing that they may be looking for is direct solutions to specific problems that they are experiencing which is not something I intend to accommodate for.

Another limitation of the solution is any particularly advanced calculation algorithm. This is in terms of potential for AI, while I intend to have a user login for saved plants, I don't intend to then keep it for saved user inputs or take any actual user details into the calculation in which it would become more similar to an AI.

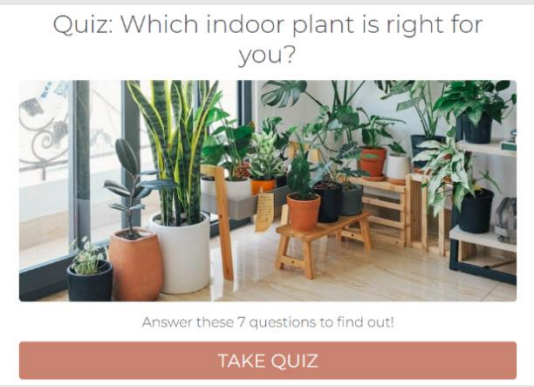
Research of Other Solutions

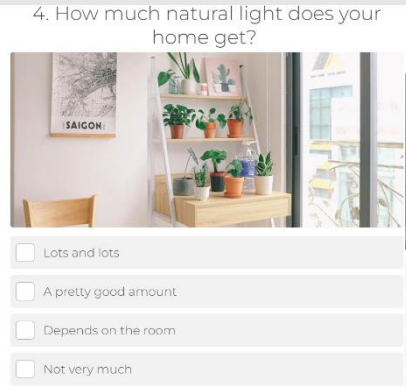
To initially research this project, I re-read a book that I had first bought when using this idea for my Extended Project Qualification. This book is called How to Grow Fresh Air and details many methods for growing and caring for plants, but most importantly it quantifies many of the attributes of plants as well as giving them an overall rating. These attributes include things like air cleansing properties, ease of care and transpiration rate. I explore this book further on in my case studies.

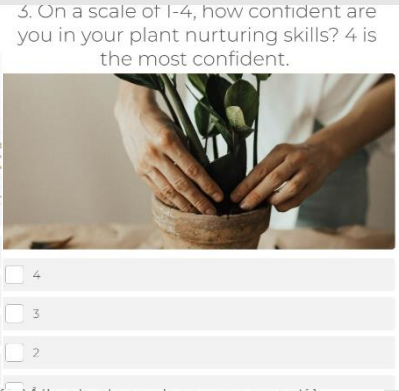
The other research I did was for the UI of my project, there are two directions I went with this, through apps and through online quizzes. In apps these were very streamlined with a huge number of different features, one notable feature was taking a picture of the location to measure light levels and while I don't intend to implement this it is an interesting idea I would explore with more time. Through quizzes I was able to see a much more simplified UI that focused far more on user retention and so it gives me more of an idea of what to do if I need to expand my criteria.

Case Study: Wholesome Culture Website

A quiz that caught my eye was on the website 'Wholesome Culture' this was because it seemed to have the cleanest GUI as well as questions that were based in empirical evidence such as asking the user about their sleep patterns or the light level in their homes and they did this by setting realistic and understandable desires the user may have.

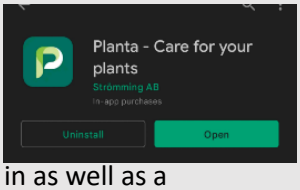




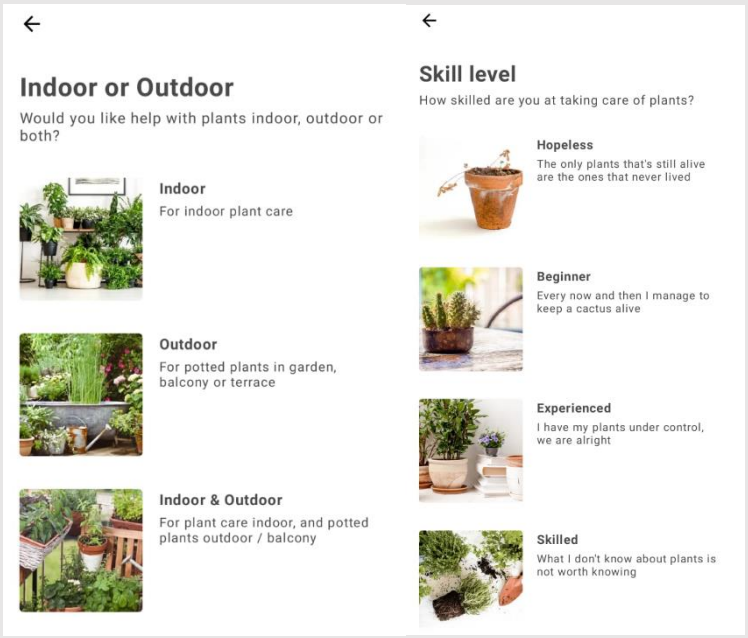


Above are the user inputs that this quiz has, you can see how the buttons are smooth and rounded as well as clearly signposted for the user. The questions are clear and make sense even if the user doesn't know much about plants but also gives options for people who do as shown by the rating of how good at plant care the user is. If the quiz assumed everyone was an expert or everyone was new to it then it wouldn't have it. The questions also come with clear pictures of what they are talking about as well as smooth and fast changing slides in order to keep the user's attention. The issues I have with this are minimal, but they involve the fact that some of the questions seemed to be incredibly shorthand/casual, not the direction I wanted to take the project in since I wanted it to not get stale quickly as the questions in this quiz often did. This case study gave me a good place to form initial project outlines, something I then built on with another case study and then my stakeholder interviews.

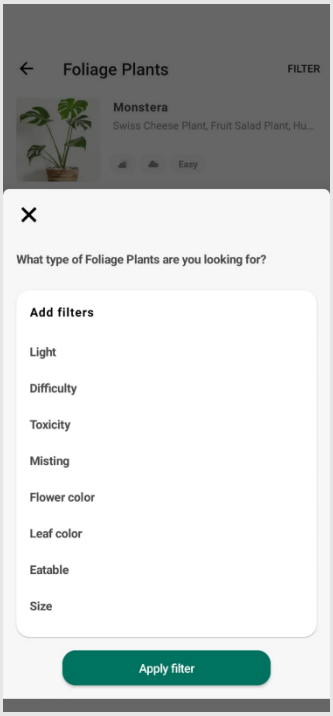
Case Study 2: Planta App



This case study was more based around finding an actual piece of software that assisted a plant owner, this was because it would be more technical. The one I chose was called Planta. This app had functions other than just recommendations such as care guides and a plant identification for users so it would build a bank of plants that the user owned. When launching the app, I was greeted with an initial sign in as well as a small set of questions very similar to the previous plant quizzes such as assessment of skill and location.

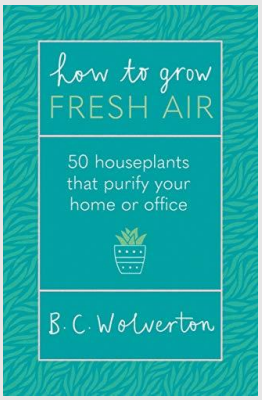


On the left are the questions I was asked. What I like about them is that they provide the user with information also about the details of the options they can select. After the initial sign up the main page of the app encourages the user to enter a plant that you own into your databank. This idea of a databank was interesting and if I had time after my first development of the plant recommendation system then I would include a user catered databank. Planta's recommendation system was unfortunately behind a paywall so I couldn't see how it was formatted or worked. At the same time though it had a filtering system on its plant search function which was very similar to what I wanted in the sense that the user specified inputs which then constructed a query to search the database. The filter here gave me ideas of the kind of restrictions I wanted to use on my plants as well as the scale I would use for it e.g., the light filter went from dark to full sun in 4 steps.

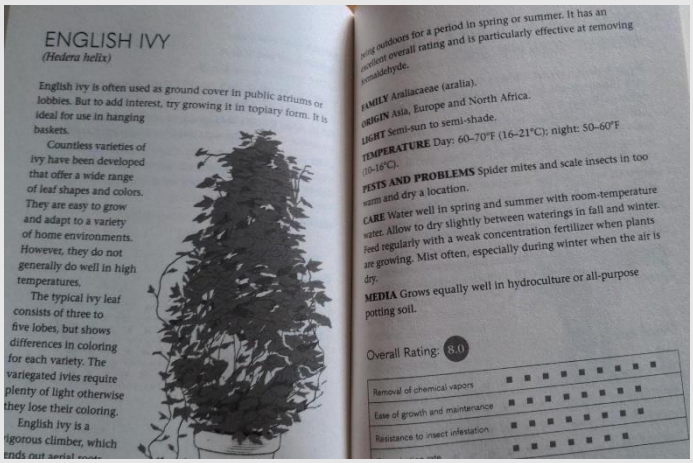


There were few things I didn't like from this app; I obviously didn't like the fact there was a paywall so I couldn't tell what the named recommendation system was. I also didn't like how large the range of filters was since there was no situation I could think of where a user would worry about misting or edibility for a houseplant. Even though there was such a large range of filters, the individual filters could have gone into more detail such as the fact there were only 3 options for ease of care, (easy, medium and hard) this felt lacking for me since there are so many components that factor into skill that there shouldn't only be 3 options. Another issue that I found with the filters is that they missed out something key that was temperature, this was also something the previous quiz had missed. My issue with this is that temperature of the room is so key for houseplants where a too cold room can cause for them to grow slowly and a too hot room can cause them to dry too fast. This goldilocks zone of temperature is different for every plant as I found in the book.

Case Study 3: How To Grow Fresh Air



This case study is from a book I found while researching how to rank plants, the basis of this book is research done by NASA compiled by an ex-researcher which contains data drawn from tests done to investigate the validity of plants in spacecraft in terms of oxygen and other chemical production. The book starts by talking about different statistics of the plants in terms of things such as oxygen concentration. The most important part of the book is the second half, this is a huge database of different plants in terms of their attributes and when I read it I realised that it was what I wanted to find my values for the plants I was going to store in my database. Although this book is from 1998 the data still very much holds up especially since the attributes I want to define my plants by are quite simple and not too scientific and will therefore not change with time and will not have inbuilt obsolescence. The data being originally from NASA made me confident in it since it gave me the reassurance that it had defined the plants well unlike Planta and the quiz which seemed to be very loose in terms of how they ranked plants.


ENGLISH IVY
(Hedera helix)

English ivy is often used as ground cover in public atriums or lobbies. But to add interest, try growing it in topiary form. It is ideal for use in hanging baskets.

Countless varieties of ivy have been developed that offer a wide range of leaf shapes and colors. They are easy to grow and adapt to a variety of home environments. However, they do not generally do well in high temperatures.

The typical ivy leaf consists of three to five lobes, but shows differences in coloring for each variety. The variegated ivies require plenty of light otherwise they lose their coloring.

English ivy is a vigorous climber, which

being outdoors for a period in spring or summer. It has an excellent overall rating and is particularly effective at removing formaldehyde.

FAMILY Araliacaeae (aralia).
ORIGIN Asia, Europe and North Africa.
LIGHT Semi-sun to semi-shade.
TEMPERATURE Day: 60–70°F (16–21°C); night: 50–60°F (10–16°C).
PESTS AND PROBLEMS Spider mites and scale insects in too warm and dry a location.
CARE Water well in spring and summer with room-temperature water. Allow to dry slightly between waterings in fall and winter. Feed regularly with a weak concentration fertilizer when plants are growing. Mist often, especially during winter when the air is dry.
MEDIA Grows equally well in hydroculture or all-purpose potting soil.

Overall Rating: 8.0

Removal of chemical vapors
Ease of growth and maintenance
Resistance to insect infestation

To the right is a picture of one of the plants in the book, English Ivy or Devil's Ivy. This has a brief introduction to the plant which contains information about its size and delves into how the plant looks, grows and works. Then on the right of it is scientific information about the plant such as its temperature range, light requirements, pests it suffers from, biological and geographical origin and care instructions. On the bottom of the page is the rating table which has stats such as ease of care, transpiration rate and other scientific information. Another thing I liked about the book was the fact that it gave every plant a numerical rating and then sorted it from highest to lowest in terms of order of appearance and it gave me an idea of how to do some of the coding for the calculations. A thing I didn't like about the book is that the plants are wired towards an office space instead of a room in a house, this means it incentivizes chemicals for productivity and prefers larger rather than smaller plants simply because they are more chemically efficient.

Coded Solutions

GUI

The most amount of variation in potential solutions to the problem is with the user interface, there are many python modules that can display windows with text and buttons, the two main ones I looked at were PyQt5 and tkinter, these were because they were the best ones to learn with little complexity. An issue with keeping the code in one language, as I wanted, was that python is not a language suited for GUI design, for instance when looking at websites they mainly use CSS for their front-end operation. Python is mostly used as an algorithmic language.

PyQt5 was a good option because it isn't specifically python, it is the blend of the QT library and python, this is a useful option because it has a lot of versatility when it comes to design and development as well as a huge variety of widgets. The versatility of design means it has a far higher ceiling where it comes to what the GUI can look like, it is very much a module you make conform to your design instead of conforming your design to it. Another advantage of it is that you can change the styles of the GUI and widgets if you don't like the default, On the left is a default button and on the right is one with the 'windows' texture. This is quite nice but at the same time it's not something I prioritise.

The other GUI module I looked at is tkinter, this is the one I had learned a bit in class so already knew my way around it. It comes preinstalled in python and so is good for a small project such as this. It is a very fast processing speed GUI because it preloads all the potential configurations on startup, this would be an issue with larger projects but with this it wouldn't because not much would be created. Another advantage of tkinter is its simple, a lot of the methods are self-explanatory and so even if I don't know something it is easy to look up since it's so widely used there are many tutorials. An issue with tkinter is that it is quite restrictive, especially when it comes to inputting images, it's also quite hard to debug and doesn't have particularly advanced widgets.

Overall, I will end up using tkinter since I have some experience with it as well as my teacher knowing how to use it, meaning he would be able to help if I ever struggled with something I didn't know the solution to. The simplicity of tkinter was also ideal for my single language project since I didn't want to stray into QT which defined classes differently to python, something that might have confused me.

Imaging

A key part of the GUI and the plant display is showing the picture of the plants recommended, there are two main ways to do this, through Binary Large Objects (BLOBs) or through simply storing images and calling them. The advantage of BLOBs is that it is most efficient when using SQL since they are just stored as a field instead of with saved images where I'd use the file directory links.

Data Storage and Retrieval

This is important since there are two ways to go about this, one is by using SQL as my database language, the advantage of this is that using SQLAlchemy as an interpreter make the implementation easy and an actual database is far more efficient and complex than anything else. For example, SQL can use the JOIN function to show the intersection between two tables which may be good if I would want to filter the results by something like pest type if the user is more experienced with a particular pest.

If I was unable to do an SQL database, I would instead use a series of lists containing objects in them. This would work well because it is simple and doesn't require any new module installs unlike SQL, keeping the overall set up the user needs to do to a minimum. A drawback of it is that it would be difficult to do a similar filter like stated before with plants and pests so it would decrease the overall complexity of the code. A python array system is what I would use if I ran out of time for the SQLAlchemy since it is more complex and I would have to learn it.

Hardware and Software Requirements

Something that is vital to know for the project is what the program needs to be run properly. This can vary depending on what I use in the code and also how intensive any calculations are. To start with are the modules that I would need to use.

If using SQLite, I would need the computer this is running on to be able to process and handle .sql files by having SQL installed as well as SQLAlchemy for the connection between database and code. SQLAlchemy is a python module that is used to create connections between a database file and the python file, it does this by treating the different tables as classes and object-oriented programming. Using SQLAlchemy I would be able to easily create and edit any databases made as well as querying them smoothly since the python file can just take the SQL commands in string format. On the right is a snippet of what the python would look like if creating a table in SQLAlchemy.

```
class Plant(Base):
    __tablename__ = 'plant'
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, unique=True, nullable=False)
    chem_remove_val = Column(Integer, nullable=False)
    ease_of_growth = Column(Integer, nullable=False)
    care_instructions = Column(String, default="")
    min_temp = Column(Integer, default=15)
    max_temp = Column(Integer, default=26)
    pests = relationship("Pest",
                         secondary=plant_pest,
                         back_populates="plants")
```

For image displaying even if not using BLOBs I would use the Python Imaging Library (PIL) which when used in tandem with a python module called Pillow would be able to manipulate and use images within the code by opening them from downloaded files. These images would probably take up the brunt of the memory in the code because even though they are compressed they still are considerably large compared to the rest of the code. Even though they take up most of the memory the total size of the code would probably not even exceed 10 MBs so any computer could run since all modern computers have enough RAM and Storage capacity to accommodate for this.

For the visuals of the code I intend to use a GUI, this will be made using tkinter since that is the python GUI module I have the most experience with as well as the fact it can keep quite concise and effective code within it and not be too complex for the actual looks of it. Unlike other GUI modules, tkinter is very much a system of GUI design that prioritizes functionality over looks. This is not to say you can't have a nice-looking program but at the same time it won't look particularly modern without having to spend a lot of time on the looks alone.

In the project file it is also imperative to have a requirements.txt file, the advantage of using my chosen IDE, PyCharm, is that it has an automatically updating requirement.txt inbuilt that keeps track of the modules the programmer uses as well as the latest version number e.g., the Pillow module for images will look like this:

```
Pillow~=9.0.0
```

Initial Specifications

From the case studies I was able to draw up some initial specifications that I would want my project to fulfil, these would then be expanded based on the stakeholder requirements.

In terms of a key, R prefix is for requirements and D is for desirables (what I achieve if I have enough time).

R1	R2	R3	R4
Create a program that recommends plants based on attributes of the user's room.	Have a user interface which takes inputs from the user and then does the calculations for the recommendation.	Have a database system which contains multiple tables which the program accesses to display.	Display the characteristics of the recommended plant and what they look like.
This is the underlying Basis for what I want the project to be at most simple.	This is the other aim I have for the project, an easy to use and simple looking GUI for the program.	This would be so I can easily represent the data being retrieved by the GUI and keep it referentially integral.	This is important because the user has to know what they are being recommended.

D1	D2	D3	D4
Add a plant saving function unique to the user potentially based on a log in.	The database is made using SQL.	Use an image storing system using BLOBs (Binary Large Objects).	Have a calculated ranking that is based off the user inputs.
This I got from Planta and it would be nice to have multiple functions of the program.	This is one of the most efficient methods of creating a database and by using SQLAlchemy (a python module) I would be able to communicate it with the main code	This is the most efficient way of storing images in a database since the image is directly loaded into the SQL in binary.	This is based off the ranking system in the book since this displayed a good numerical measure of how good a plant was.

Stakeholders

Establishing Stakeholders

The clients for this software range in huge ways, from plant experts looking for reassurance with how to care for their plants, to prospective plant owners looking to start owning houseplants and are in desperate need of advice.

In order to accommodate for this my chosen stakeholders are taking one individual who has never owned a houseplant before as well as someone who has amassed a diverse collection but still needs some tips to stop them from dying as seasons change.

The most important aspect for testing if the software works is if a recommended plant and position actually survives and thrives according to the program. The reason this is convenient is that one of my stakeholders, Erin Campbell, lives in an attic room at the top of her house so all attributes of the room are easily controlled such as the temperature, wind flow and light levels due to the light coming from very controllable sources. Another advantage of Erin is that she already owns some houseplants so is quite experienced on the topic.

The convenience in the other stakeholder, Matthew Dodd, is that he does not know much about houseplants, which then tests the other vital part of the software, accessibility. By having someone who doesn't know much about the terminology or the methods, the program will have to be understandable to Matthew.

Creating Questions

Since both of my stakeholders are quite different in terms of plant experience and technical knowledge, by asking the same questions and evaluating their response I can get a better spread of how they would react to the program instead of tailoring questions for them especially since I want the program to work irrespective of skill or experience. In order to do this, I have constructed a script that I will then ask my

stakeholder in a recorded interview. These questions have to be a mix of general feelings about software and plants and then more specific according to how I vaguely want to do the program.

QID	Question	Reason
Q1	How well do you know how your room can cater to plants?	This will give me the basis of how in depth my user inputs would be.
Q2	How good are you at looking after plants?	This would assure me of the range of ability of the stakeholders
Q3	What factors in plants are important to you?	This would help me refine what the options would be other than just the stats of their room.
Q4	Do you prefer a more efficient program or a better looking one?	It would be more efficient to have no graphics and just do it off command line.
Q5	How much information do you want to see about a plant?	This would give me understanding of what I would display to users.

Interviewing Stakeholders

Below are the videos of me interviewing my stakeholders, on the left is Erin and on the right is Matthew.

Erin's Interview

Matthews' Interview

From these I can establish what they want and move onto finalizing my specifications for design.

Success Criteria

After interviewing the stakeholders, I am able to make a verdict on what my success criteria will be while containing more technical information about inputs and attributes shown by the GUI. For example, Matthew said that it mattered to him what the plants looked like and what size they are, therefore I would want to show a picture of the plant and also give information about its size.

R1	R2	R3	R4	R5
Create a program that recommends plants based on attributes of the user's room.	Have a user interface which takes inputs from the user and then does the calculations for the recommendation.	Have a database system which contains multiple tables which the program accesses to display.	Display the characteristics of the recommended plant and what they look like.	Characteristics shown include things like temperature, ease of maintenance, care instructions, size and what the plant looks like.
This is the underlying Basis for what I want the project to be at most simple.	This is the other aim I have for the project, an easy to use and simple looking GUI for the program.	This would be so I can easily represent the data being retrieved by the GUI and keep it referentially integral.	This is important because the user has to know what they are being recommended.	These are the things that Matthew and Erin asked to be displayed from the interview.

D1	D2	D3	D4	D5
Add a plant saving function unique to the user potentially based on a log in.	The database is made using SQL.	Use an image storing system using BLOBs (Binary Large Objects).	Have a calculated ranking that is based off the user inputs.	Whilst needing to look good the GUI also must be simple enough to understand in order to be appropriately efficient.
This I got from Planta and it would be nice to have multiple functions of the program.	This is one of the most efficient methods of creating a database and by using SQLAlchemy (a python module) I would be able to communicate it with the main code	This is the most efficient way of storing images in a database since the image is directly loaded into the SQL in binary.	This is based off the ranking system in the book since this displayed a good numerical measure of how good a plant was.	This is a balance of what both my stakeholders said about the system, whilst a in console program would be more efficient than a GUI, the GUI still has to be there, therefore the GUI has to be easy to understand so it's fast as possible for the user.

Design

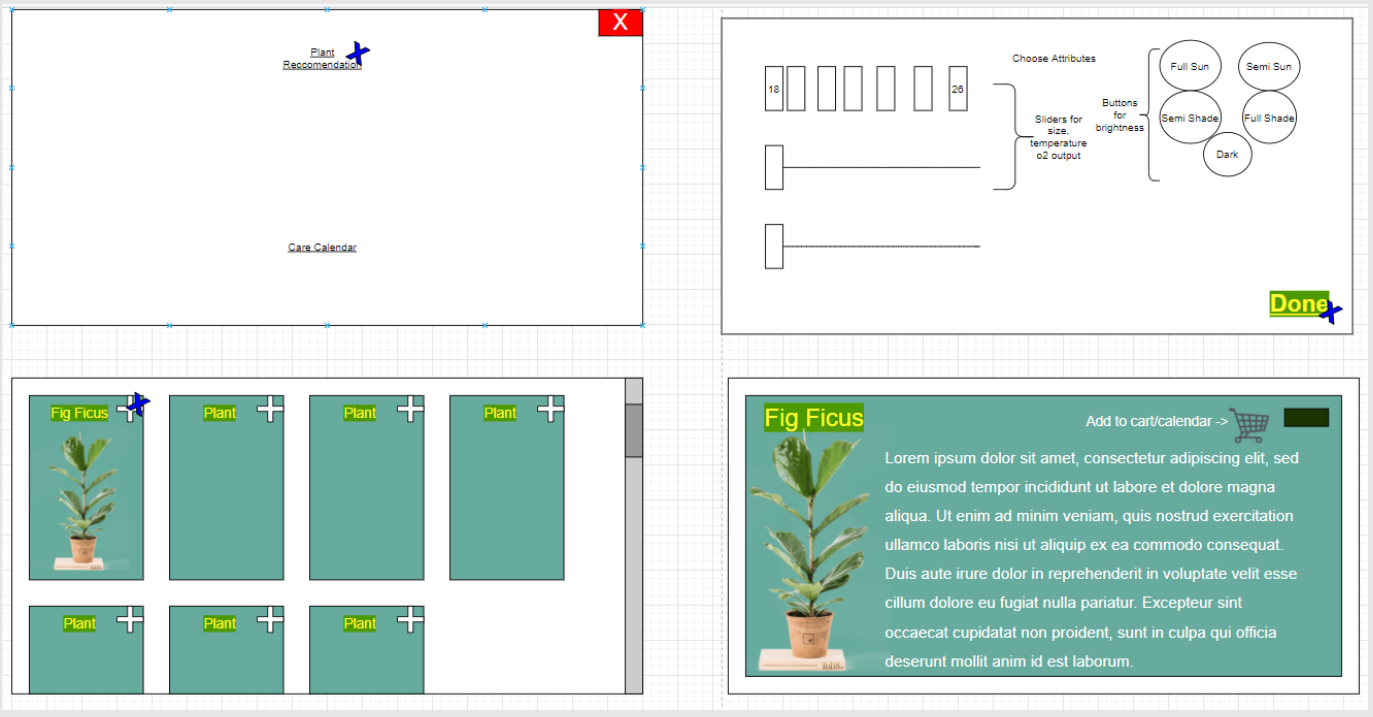
For this project I have to deconstruct it into separate parts in order for it to be easier to code and manage, this would allow me to abstract many parts of the code since I only need to put in plant details into the database side. So, by splitting that from the rest of the code I don't have to worry about it into I implement them together. Because of this I have deconstructed the project into three major components, the algorithm, the database and the GUI. Within these I have also further deconstructed, like for each attribute there is a different algorithm to calculate its ranking and there are many different sub-routines within the GUI. The different sub-routines within the GUI make the code far more modular instead of running one large script, this makes the code easier to debug and test since I would be able to simply run individual parts of the GUI to see what they looked like without having to worry about implementing them with the rest of the GUI.

GUI

First Design

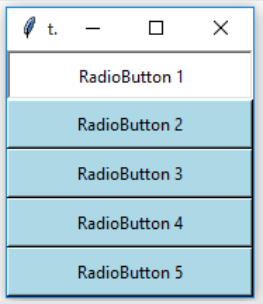
I started to design my GUI so I will know what I need from the processing code.

I did this by initially creating a design in an online program called Draw.io which I had previously used to construct the database diagram as well as some flowcharts for a previous task. It was convenient because I was able to make a simplistic layout of the GUI without having to code anything.



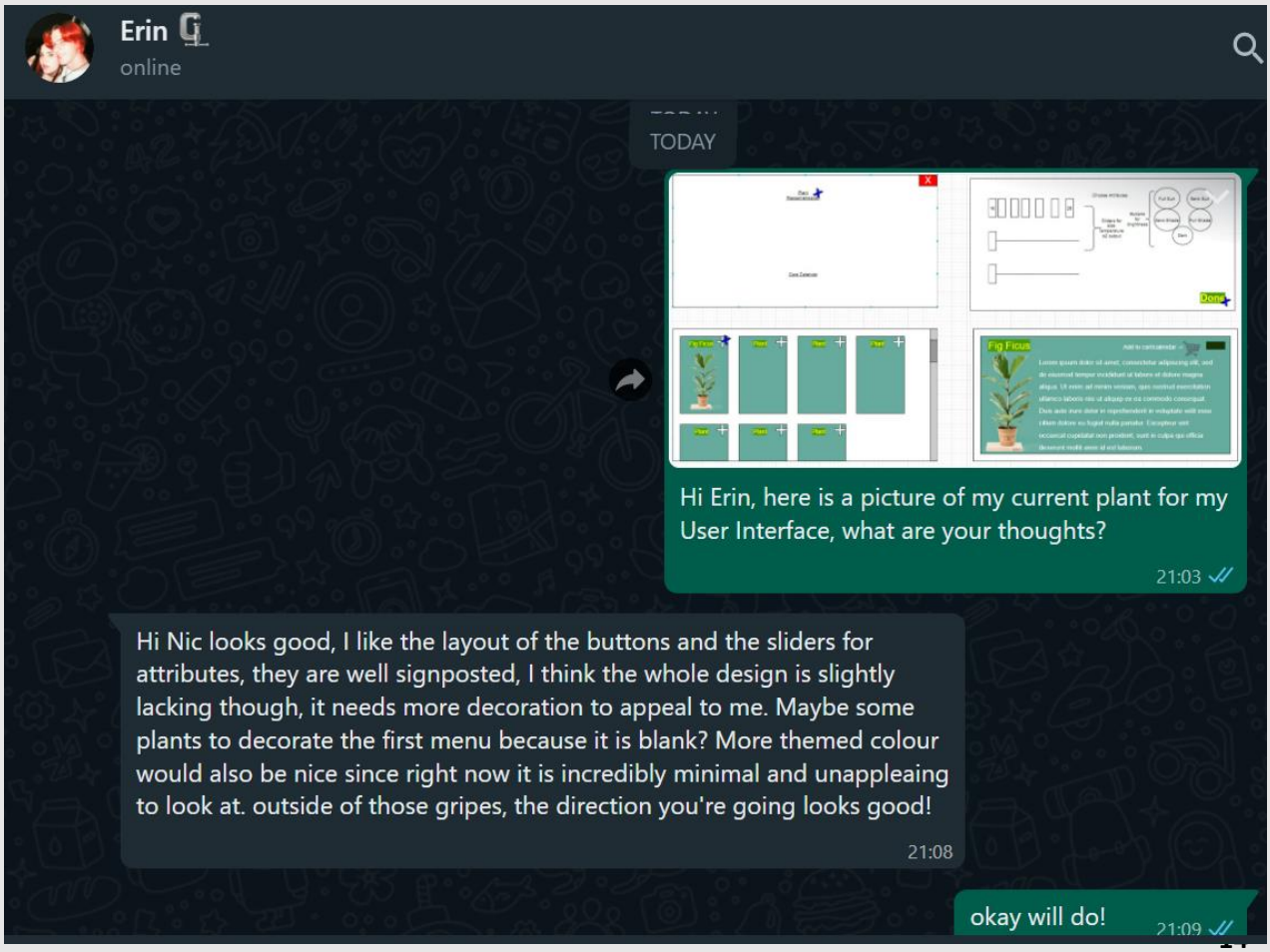
The draw.io (above) is a good design because it is clean and simple, it also showcases how the screen could look when expanding a plant to see its statistics. This is a nice idea because the user can then view many plants at the same time if they don't like the first one. On the right of the plant box frame is a scroll wheel, this is quite intuitive design since it is universally known how to use it; I would only include a scroll wheel if the list of plants being displayed was long enough. The colours for the plant box are how

I would like to colour scheme my entire code since it is very naturalistic and fits with the themes of plants and greenery. On the first frame is the main menu, from this the user can select either off the program, get recommended a plant or go onto the saved user's plants, known as the care calendar since this also tells the user when to water and care for the plants. The user inputs on the second slide are split into 3 different types, buttons of a range, buttons of a selection, and sliders. The buttons for a range are the temperature buttons which stem from 18 to 26 °C, the buttons for selection are for brightness since the data for those is discrete so it can't be expressed very well with a slider. The sliders are for continuous data such as the ability the user has for plant care and the height if the user should want to give an actual measurement. I like this layout because it is self-explanatory and also keeps the tasks the user has to do concise for the screen. The expanded box for the plant is the final frame, it shows an enlarged picture of the plant as well as plenty of information about the plant, for this design the information shown is the basic lorum ipsum but in the actual information would be shown for the final design. For the individual buttons I would like to use something called radio buttons since they stay pressed down after being pressed so the user knows what they've selected, as seen on the right.



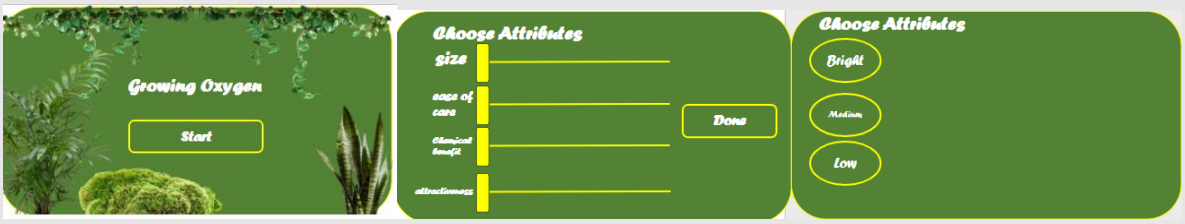
Stakeholder Response

After this initial design of the GUI, I got into contact with Erin, one of my stakeholders, I chose to ask Erin about the GUI rather than Matthew because she has a more artistic and creative viewpoint than him. I started by presenting the image to her, above is the following text conversation.

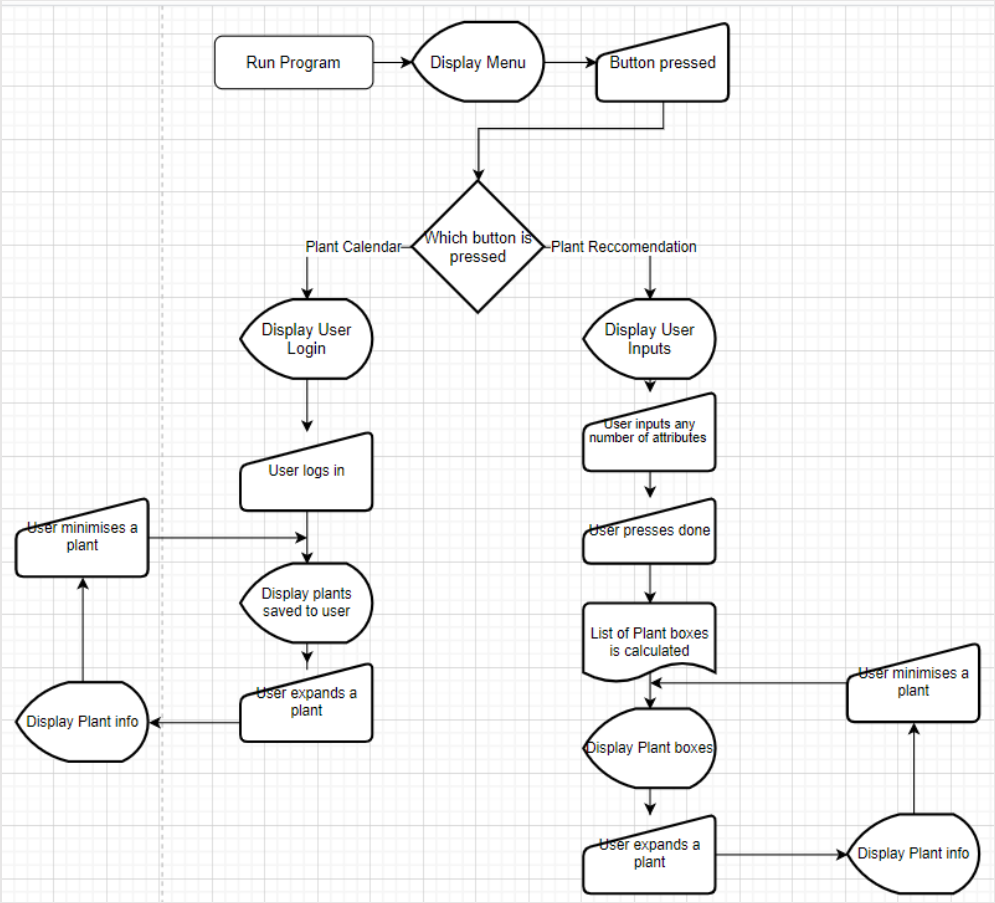


Second Design

Below is a more stylized version of the first design, it is similar except it has rounded edges and circular buttons, this is according to Erin's feedback as seen with the deeper green background. This is also what a commercialized version of the program would look like. It leaves appropriate space to each attribute selection, even splitting them up into multiple sections and frames. There is also a far more attractive main menu slide which is adorned with plants, this is quite unrealistic to achieve because there is a lot of difficulty in precisely placing images to look a specific way in tkinter. This is a far more long-term design and that is why it isn't particularly fleshed out, but I think it is good to have a concept of the long-term projection of the program.



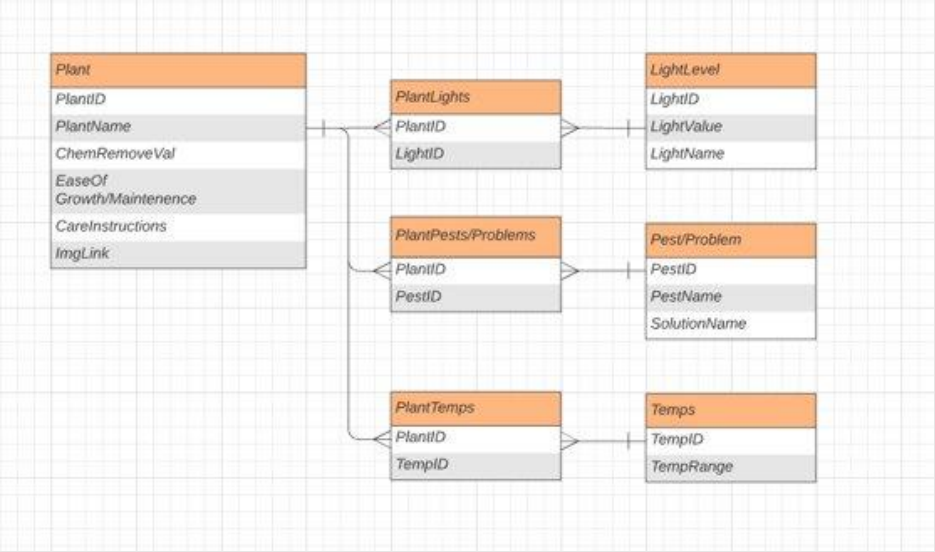
GUI Flowchart



Left is a flowchart of what the GUI should do if started. This splits into two depending on what option is selected, the loops at the bottom are intended to be incredibly similar in order to keep the program simple and practice modular code with sub routines.

Database

Initial Design

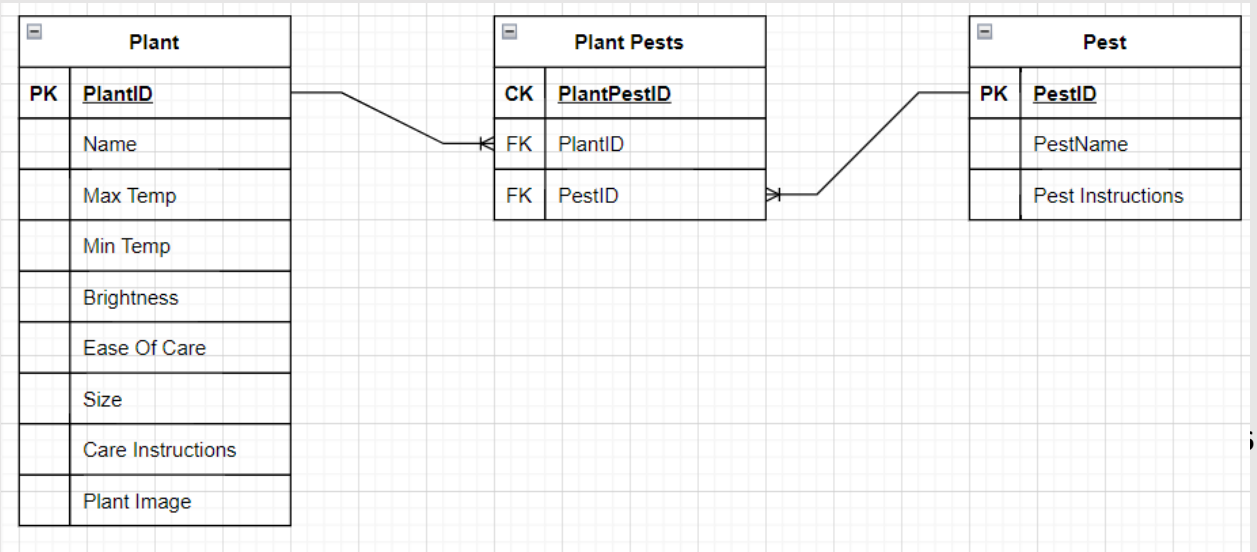


The first part of my design was to construct an initial database, this was in order to understand what my later coding elements would be since the database is the most unchanging part of the system.

I did this by using the book mentioned before, *How to Grow Fresh Air by* B.C. Wolverton since it essentially contains a database of 50 plants with corresponding attributes and diagrams. From this I was able to decide on my plant attributes as: Name, Rate of chemical vapor removal, ease of growth, care instructions, ideal light level, ideal temperature and common pests and problems.

Final Design

Whilst actually coding up the code for the database I was told by my teacher that my database could be improved due to the fact I had too many linked lists with no point, the temperature could just be expressed by a max and min attribute to indicate the range the plant can survive at. The light level could be a simple numerical value that associates with a dictionary of names, e.g., light level attribute of 5 connects to 'Full Sun'. However, whilst pests use a linked list, it would stay like that in order for the person filling the list not having to write out the solution instructions each time. Due to this change in database the final design for my SQL database looked like this:

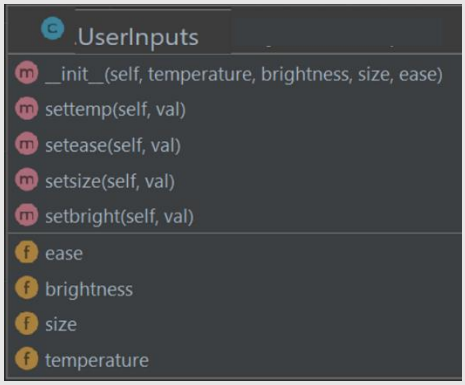


Algorithm

The main aims of designing for the algorithm is knowing what I want it to do, I want it to read user inputs from the GUI and then retrieve a list of plants that matches these inputs from the database. Therefore, there are three stages of this, managing the user inputs, ranking the plants according to these inputs and then sending it into the GUI in a manageable sense.

User Inputs

The user inputs would essentially be a list of variables, in order to store them and easily get them in one place I would use a class. This would have an attribute each aligning with the user inputs, temperature, size, brightness and ease, each of these attributes would be public with a *set_attribute(val)* method that is called by a button in the GUI. The class diagram of the User Inputs would look like this:

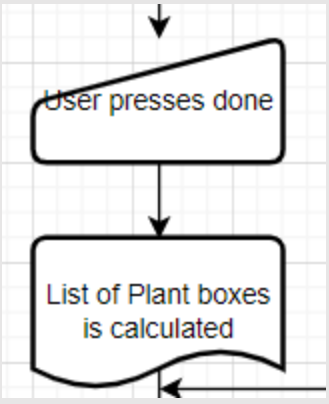


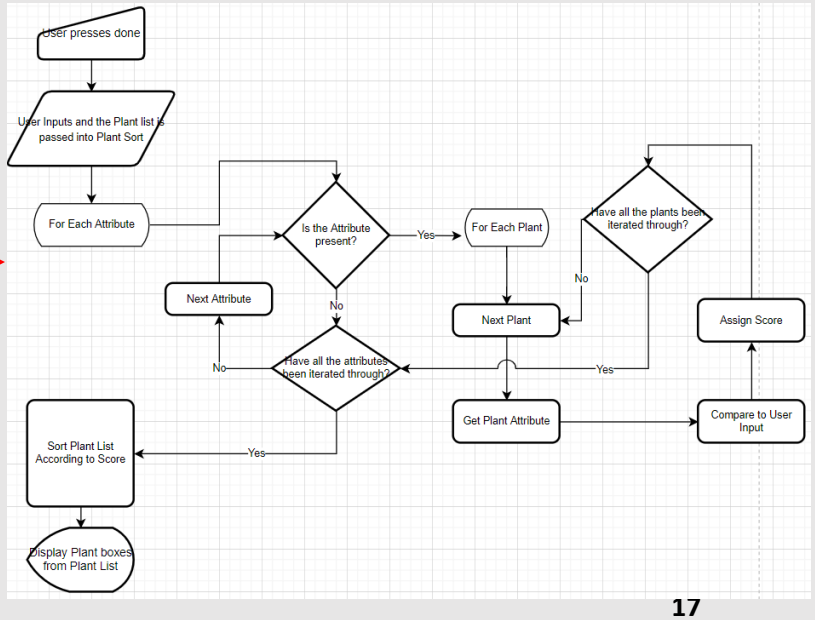
It would take an *__init__()* function that initialized all the attributes at first with None and then as buttons were pressed on the GUI an object of the User Inputs type would update with the set methods.

Calculating Ranking

After the 'Done' button has been pressed on the User Inputs slide of the GUI, a list of plants has to then be passed into the GUI so it can show the plant boxes in a correct order, this order will be calculated according to an object input from the User Inputs. Therefore, I need to create a function that takes User Inputs, and a series of plant objects and calculates their position in order for the GUI to read it.

Below is the section of the GUI flowchart updated with where the calculations take place, this process will be explained earlier on.

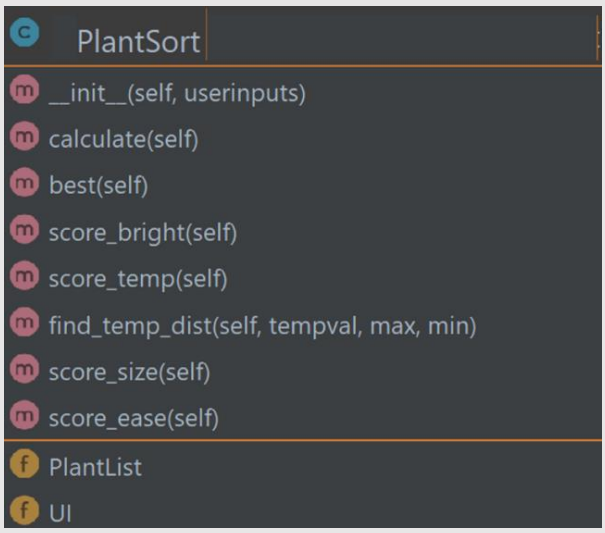




I needed to be able to sort the plant list according to what was best in order to ensure that a plethora of plants came up. To start doing this I had to change the Plant table to include a numerical value known as *desirability.score.* This score would be set to 0 when the field is first created and then be called inside of the calculations class when the score needed be changed, updating its value in the table. I.e., depending on how close the user input was to the attribute of the current selected plant. At the end of the final calculation the plant list would be sorted according to the desirability score.

Plant	
PK	PlantID
	Name
	Max Temp
	Min Temp
	Brightness
	Ease Of Care
	Size
	Care Instructions
	Plant Image
	Desireability Score

The best way of doing this is again with a class called PlantSort with a constructor of User Inputs and the default Plant List because it would be able to change the desirability scores of the plants using methods. It would iterate through each plant in the list and according to its attribute's proximity to the user inputs' attribute, it would assign a score. For example, if the user input temperature is within the plant's range of temperatures it would assign a score of 5, these calculations are explained more in depth later on. The class diagram for plant sort would end up looking like this:



All the individual methods for finding the scores are quite similar such as *score_bright()* which finds the desirability score for the plants depending on their brightness. The only method different to this set layout is the *score_temp()* method since that operates using a range of values. The *find_temp_dist()* method would return the distance the range of plant temperature is to the user input temperature which score temp would then score with. This is because for the other attributes they are in the same format for both the user input and the plant attribute so finding how close they are to the desired value is easy. Especially since they are all passed in as a numerical value. After the calculate method has been run the *PlantList* attribute will be the sorted plant list which is then called by the GUI.

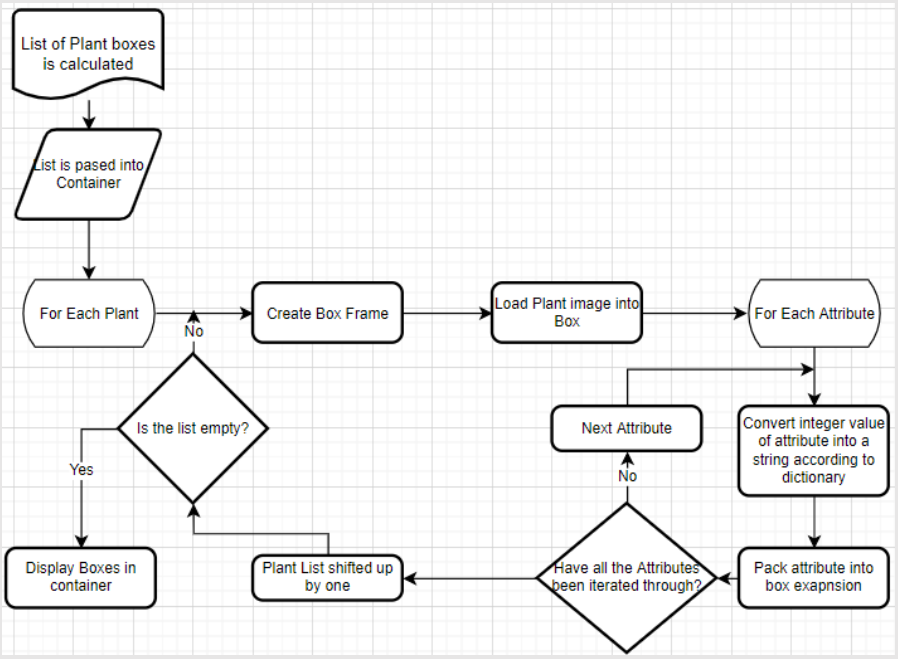
GUI Inputs

The next function the algorithm does is pass into the GUI for the plant boxes, the plant boxes are made by looping through the plant list and creating a Box according to the Plant passed into it. This turns the plant list into a priority queue, with priority ordered by the score, the GUI takes things off the queue starting from the first item, showing the FIFO nature of the queue. An example of the queue would be on the left and on the right is when the first Plant Box is initialized.

Position	Plant	Score
0	Devil's Ivy	20
1	Snake Plant	15
2	Peace Lily	12
3	Orchid	8

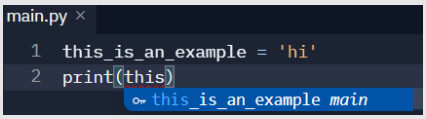
Position	Plant	Score
0	Snake Plant	15
1	Peace Lily	12
2	Orchid	8
3	Null	Null

These boxes would be initialized inside of a plant container frame, which would operate as seen in the diagram below.

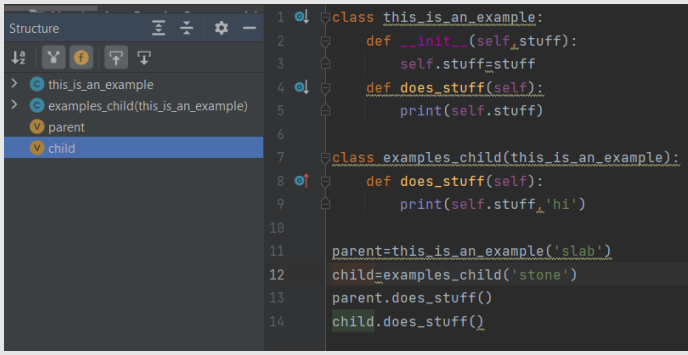


The IDE

An important part of designing and starting a project is choosing what to code it in, for this project I have two IDEs (intergrated development environment) I plan to use one is an online IDE called replit.com. This is a useful IDE because it is very self contained, when running a GUI, instead of a window popping up, the tab in the browser just displays it alongside the code and the console, meaning that coding up a GUI is far easier to test. Another advantage of replit includes automatic online storage and saving as well as something called 'coding intelligence' where it finishes statements as you start them, an example is to the right. Whilst this isn't unique to replit, it is a distinct advantage over python's default IDLE. Another advantage of replit is that it is fully online so the saves and launch are incredibly easy especially since for the start of this project I was using the school computers which struggle with any other IDE since many of them don't even have python installed.



The other IDE is called PyCharm, this is a very useful IDE because it does everything that replit does and more. The biggest advantage of it is that it can implement with github so I can save my code on github and then bring it onto PyCharm, this means that the online integration and saving still is viable. Another advantage of it is the database implmentation, the user can view and edit the databse while the code is running. There is also the advantage of variable tracking which makes running the code very easy since I can put in a break point and check the different variables to see if they are running correctly. There are also many visualisation perks of PyCharm such as automatically generated class and database diagrams so the user can see how their system interacts. As seen on the right the code keeps track of the varaiables and classes of the code.



Data Structures

For the code I will use a multitude of different data structures, some of which I will create myself using object oriented design and some of which are classes provided by the modules I will be using. Below is a table entailing the new objects being used and then each data structure planned for the code as well as what part of the code it will be used for.

Name	Data Type	Location in Code	Function	Notable Atributes/Methods
Tk instance	Class	GUI	Creates a main container and window for the GUI.	• self.title – puts a title for the window • self.resizeable – defines what dimensions the window is resizeable for
PlantApp	Tk instance	GUI	Acts as the main container for the GUI and has master over all the windows within it.	• self.frames – dictionary containing indexes for different windows to be accessed by the plant app. • self.show_frame() – takes a frame to show and packs it into the PlantApp container • self.sort_PlantList() – sorts the plant list to repack into the PlantContainer frame
Tk frame	Class	GUI	Creates a frame to be stored in a container.	• self.screenwidth/height – establishes how much room the frame has to work with
Tk button	Class	GUI	Creates a button to pack into a container.	• self.text – the text displayed on the button • self.command – what the button does when it is pressed
Tk label	Class	GUI	Creates a text box within the frame.	• self.text – the text displayed
Menu	Tk frame	GUI (PlantApp)	Creates an initial menu for the user to select plant reccomendation or calendar.	• self.create_widgets() – creates the buttons and labels packed within the frame of the menu
self.plant_reccomendation_choice	Tk button	GUI (Menu)	Sits in the menu frame and shows the next frame of the GUI when pressed.	• self.master.show_frame() – goes through to the Plant App container to show the next frame of the GUI
UserInputsMenu	Tk frame	GUI (PlantApp)	Holds all the widgets for the user inputs about their room.	• self.plant_reccomendation() – sets up all the labels, buttons and sliders to be packed in the frame

Tk radio button	Class	GUI (UserInputsMenu)	Similar to the button but if one of these is clicked then all the other buttons connected to it are marked as unclicked.	• self.value – indexes it's lable within the series of radio buttons • self.variable – defines the variable that joins a series of buttons together • self.command – same as with a Tk button
Tk scale	Class	GUI	Creates a scrollable slider which can set values depending on how far has been scrolled.	• self.from/to – defines the range of the scale • self.command – what finction is excecuted when the value of the scale changes
self.tempbutton1	Tk radio button	GUI (UserInputsMenu)	Sets the user input temperature to 18.	• GOxygen.settemp(18) – sets the temperature to 18.
self.easeslider	Tk scale	GUI (UserInputsMenu)	Slider for the ease of maintinence.	• self.setease() – automatically passes the value of the sldier into a GOxygen function
self.DoneButton	Tk button	GUI (UserInputsMenu)	The button the user presses when done with the user inputs.	• self.attributesdone() – excecutes self.sort_PlantList from the master according to the Goxygen values, then shows the next frame of the GUI
PlantContainer	Tk frame	GUI (PlantApp)	A container to hold all the plant boxes.	• self.frames – list of all the plant boxes for the container to be iterated through and packed in.
PlantBox	Tk frame	GUI (PlantContainer)	A box to display the attributes of an individula plant and switch between the more or less information of it.	• self.frames – contains the first and second states of the plant box • self.show_frame() –switches between which frame is being currently shown.
BoxPhase1	Tk frame	GUI (PlantBox)	The first phase of the plant box that just displays the plant name and what it looks like.	• self.imagelabel – displays the image of the particular plant by packing it into a label. • self.boxtitle – shows the name of the plant being displayed
self.expandbutton	Tk button	GUI (BoxPahse1)	When pressed, show the next phase of the plant box.	• self.expand() – forgets the current frame and shows the next frame of the box
BoxPhase2	Tk frame	GUI (PlantBox)	The second phase of the plant box that shows more information about the plant.	• self.imagelabel – same as with box phase 1 • self.textbox – shows all the attirbutes of the plant by retireveing the values of the plant from some dictionaries

self.minimisebutton	Tk button	GUI (BoxPahse2)	When pressed, show the previous phase of the plant box.	• self.expand() – forgets the current frame and shows the previous frame of the box
Plant	Class	Database	Sets up a temproary plant object that takes various attributes while the database is still being made.	• self.name – the name of the plant • self.pests – the object that contains a list of pests • self.desirebability_score – the deisreability score of the plant • self.scorechange() – changes the desireability score by whatever value is passed into it
Pests	Class	Database (Plant)	Sets up the pes tlist for a given plant according to a dictionary.	• self.pestlist – sets up a pest list depedning on values passed into the constructor and if there are no values, satys empty
pest_dict	Dictionary	Database (Pests)	Outlines the different pests so only index values have to be passed into pests.	•
PlantList	List	Databse	Sets up a list of the different plants consisting of a lot of plant objects.	•
UserInputs	Class	Algorithm	Holds the values for the user inputs that can be changed by the GUI.	• self.temperature – the inputted temperature of the user object • self.settemp() – changes the temperature attribute with the passed value
GOxygen	UserInputs	GUI/Algorithm	An initialising of user inputs that the GUI can access and change.	• Initially made with all values as None since the user may not want to fill it all in
PlantSort	Class	Algorithm	Takes userinputs as a parameter and then compares them to the values of PlantList and scores appropriately.	• self.calculate() – runs through all the scoring procedures and then sorts by desireability score • self.best() – returns the first value of the plant list after sorting in order to easily test the calculation
self.score_bright	Function	Algoritihm (PlantSort)	Iterates through the plant list, comparing each plants brightness value with the user inputs' brightness value.	• The individual calculations are talked about in the development section
app	PlantApp	GUI	An initialisation of the PlantApp class for the code to run off.	• After creating the app variable, an app.mainloop() command starts the program running

Proposed Tests

In order to make sure my code runs properly without errors I will need to test it and change what the code entails according to the result of the test, this is because I don't expect it to work first time doing perfectly what I want. The method of testing for the GUI, algorithm and database are all different so I need to split it up into the three sections again.

GUI Tests

There are two main ways that I'll change my GUI after testing it, the first is if the code does what I want it to which I can see by running the GUI and just seeing if it all works as intended. The second is stylistic testing where I actually see whether I want the GUI to have the shading, font display and layout I want it to.

For the first part of this it is checking if the code works so if I run the code and the frames have initialized wrong or the plants aren't displaying correctly, whilst I may not know what the solution is. It is quite easy to identify the problem.

For stylistic tests, it is a lot vaguer, whilst it could be something simple like what grid resolution is best for the plant boxes, it may be something more abstract like the colour of the text boxes not matching the colour of the background. In order to test things like this I would need to iterate multiple times with slight changes to the GUI, this is the advantage of the code being so modular because it means I can run individual frames without needing to run the rest of the code or affect it in any way.

Algorithm Tests

The tests for the algorithm are probably the most testing I'm going to do for the project. There are three main ways of testing for a given function that I will use. Firstly is an initial run to make sure it runs and doesn't error out, if it does then I obviously have to change it just so it doesn't interrupt the flow of the code. Second is a test to see if it can take an input and outputs anything, the way this will be done is see if the returned value is the correct type of data structure. Since the tests are going to be built using an inbuilt module called unittest, I will be using a checking method called *assertIsInstance()* where it checks if a given parameter is of the correct type whether that be an integer, an array or even a constructed class. Finally, there are tests to make sure that the algorithm calculates values properly, this will be done by me calculating what the values should be according to the algorithm and then running an *assertIsEqual()* l function on the result from the code and comparing it to my result. Any test failure would mean I have to reiterate my code and change what it does so it performs correctly.

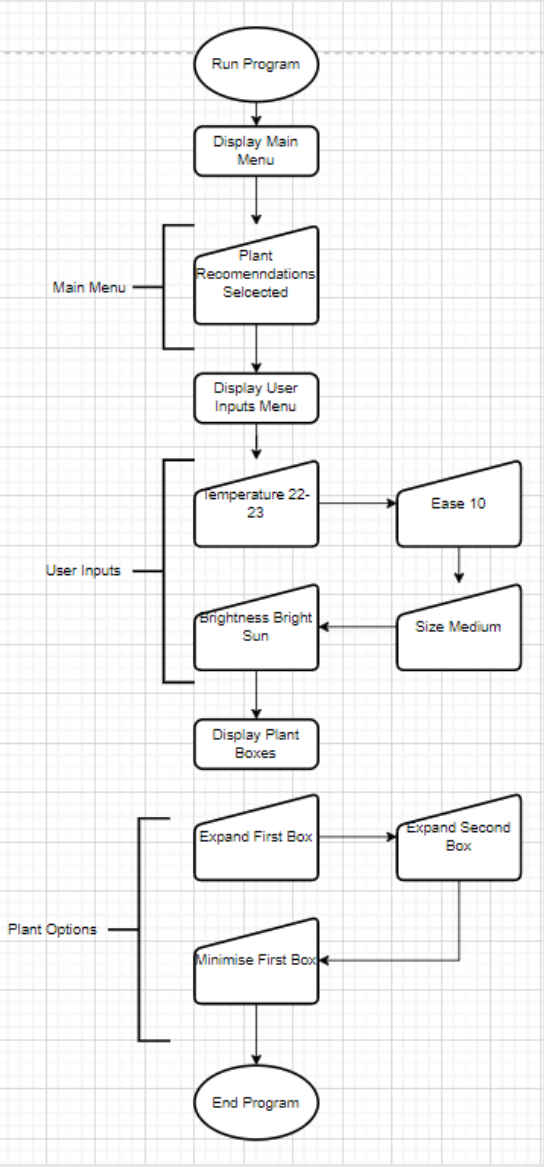
Database Tests

In order to test the database there are two aspects of it, ensure that the tables are set up correctly and then make sure that the rest of the code is able to communicate with the database. An advantage of using the PyCharm IDE is that it can display diagrams for data structures so for the database it will be able to show me the link between tables so if the database is assembled correctly then I will know. Another way of doing it is by using some placeholder values for the database and then running some example queries to see if the tables interact with each other correctly e.g., if a primary key is actually a composite foreign key. To test if the code interacts with the database correctly, I can run some unittest

tests and check if the values recognized by the code are correct with what the values in the tables are, for example testing if the first element is recognized as ‘devils ivy’ by using *self.assertIsEqual(self.name,‘devils ivy’)*.

Post-Development Test

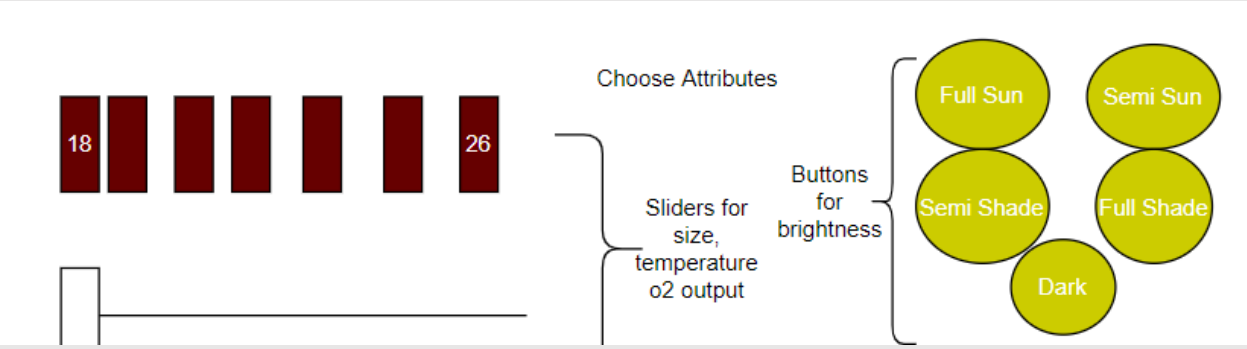
This test is designed to simulate a user experience of the system, so by calculating what the response will be myself I can compare it with the result of the code. This test will be used in my evaluation in order to see if a simulated user will be able to fully use the code. Below is a flowchart of what the inputs will be. When I do the test I will also keep track of the various scores to be calculated so I can also double check all the desirability scores are correct. I can do this by tracking the variables as they change when I run the code in the python console, this is another advantage of PyCharm.



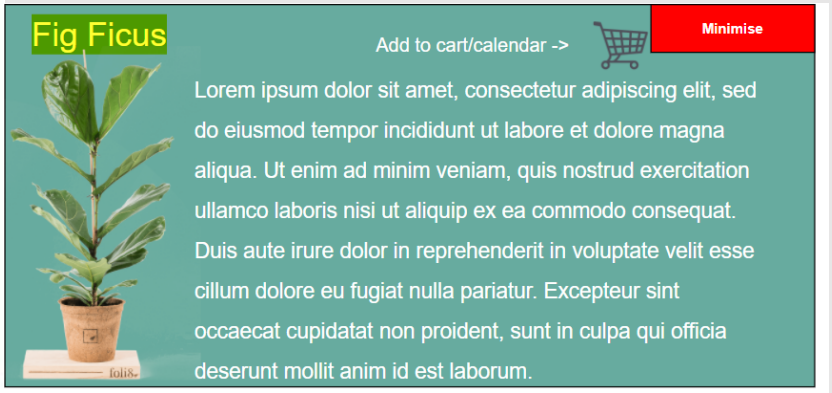
Usability

A big requirement of a program like this is the usability of the software. This entails things like how easy the program is to understand, how easy it is to use how error prone the program is and how engaging it is for the user.

A way to achieve this is by brightly and distinctly colour coding the different features of the GUI, this is good because it means the user can tell different elements apart in order to click them. At the same time the text boxes will be highlighted in well contrasted colours. By doing this I am able to get good stylistic colour pallets that are still understandable. An example below is for the temperature and brightness buttons and how they are contrasted with a dark background and white text.



Since I want to be able to display the recommended plant information as stated in R4 of my success criteria, the option to show it has to be visible whilst still being stylistically attractive, to do this I can make the buttons for expansion and minimizing obvious and keep the example text clear. As seen below with an example of the plant box with a highlighted expand and minimize button. The fact that the buttons are green and red is good because that is the universal sign for on and off, so it makes sense to have it so even those who can't read well can see that they oppose each other so it becomes quite a clear back button.



Another requirement of usability features is that the program is engaging, in order to ensure this I need to make the GUI design interesting and stylistically good as well as functional and clear like above. This is again done with things like what I said in R4 of my success criteria where I state that the image of the plant be shown. The fact that the image of the plant is shown makes the GUI more attractive and engaging for the user which is needed for the program to even be good.

The final usability requirement is for an error tolerant code, this can be done by ensuring none of my pages lead to a dead end and any inputs the user enters is recognized by the code, this means that any errors experienced by the user are not caused by them but instead caused by the code itself. The user inputs menu can do this by taking discrete data values and inputs. Since the input for the ease of care is a slider, by making sure the data values are discrete it will make the calculations a lot easier since they will be integer values 1-10 instead of potentially something like 4.5 which would make the desirability score input a float value.

When evaluating the code, I can measure how usable the code is by seeing what happens when my stakeholder uses it. This means that I will get a video of Erin running through the code and evaluate how she enjoys the GUI and program in order to see how engaging and clear it is. To measure things like ease of understanding I'll have to give it to many different individuals to test how different people find the program. For things like error tolerance, I can measure it using my post-development test and other unit testing in order to form a wide view of what can be inputted and what can't.

Development Process

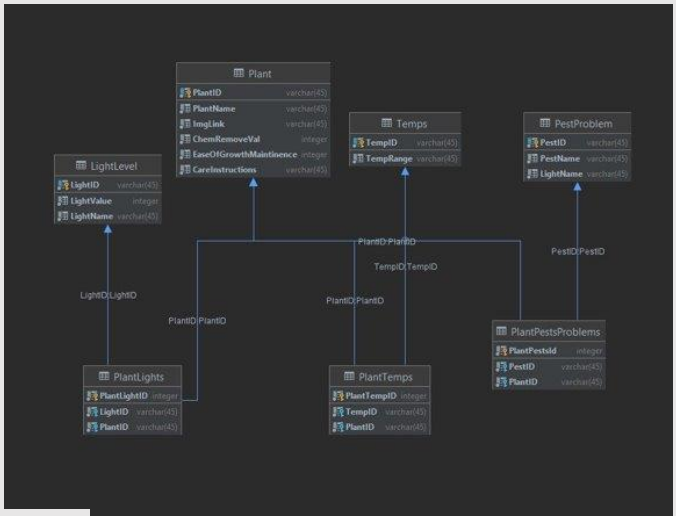
Databases

I started by constructing the designed database using an online creator called DBSQL which is a website where one replicates a database diagram in MYSQL and then it gives the code used to make the diagram into an actual database, this was extremely convenient because it gave me a baseline for how I wanted my final database to look. Shown below is how the diagram looked in DBSQL.

Following this I imported the code into PyCharm and into my main project file from GitHub, but I soon realised the issue with my use of DBSQL which was the fact that the database created was in MySQL and not in SQLite, the optimal formatting for integrating with python, luckily the two formats are very similar so all I had to do was change the code in an online IDE from MySQL to SQLite. As shown by these two pictures coding for the same table, PlantLights. On the left is it in MYSQL and even though they are similar pieces of code, the defining of constraints was something I was not familiar with, which is why I then converted them into SQLite as seen on the right.

```
CREATE TABLE `PlantLights`
(
 `LightID`      varchar(45) NOT NULL ,
 `PlantID`      varchar(45) NOT NULL ,
 `PLantLightID` integer NOT NULL AUTO_INCREMENT ,

PRIMARY KEY (`PLantLightID`),
KEY `fkIdx_65` (`LightID`),
CONSTRAINT `FK_63` FOREIGN KEY `fkIdx_65` (`LightID`) REFERENCES `Light
KEY `fkIdx_68` (`PlantID`),
CONSTRAINT `FK_66` FOREIGN KEY `fkIdx_68` (`PlantID`) REFERENCES `Plant
);
```

```
CREATE TABLE PlantLights
(
 LightID      VARCHAR(45) NOT NULL ,
 PlantID      VARCHAR(45) NOT NULL ,
 PlantLightID INTEGER NOT NULL  PRIMARY KEY,
FOREIGN KEY (LightID) REFERENCES LightLevel (LightID),
FOREIGN KEY (PlantID) REFERENCES Plant (PlantID)
);
```



Converting to SQLite made the database work inside of PyCharm meaning I was able to create a sufficient database diagram as seen to the left. Unfortunately, after creating this, I was told that it would be easier to create and manipulate the whole database in SQLAlchemy, the integrator I was using the manipulate the data in python, than by bringing the data into the SQLAlchemy module, manipulating it and then sending it to the main python code. Whilst telling me this my teacher also let me know that some parts of my database were sloppy and I did not need as many link tables. I subsequently used a template for my new database which I had used previously in my A-Level and produced what is shown below in SQLAlchemy. On the left is my base tables being made and on the right is my link table that joins the two together.

After reaching this point of my database creation I decided to leave it for a while and start to work on my other parts of code in order to know what my calculations will require as well as needing a break from the rest of my code.

```
class Plant(Base):
    __tablename__ = 'plant'
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, unique=True, nullable=False)
    chem_remove_val = Column(Integer, nullable=False)
    ease_of_growth = Column(Integer, nullable=False)
    care_instructions = Column(String, default="")
    min_temp = Column(Integer, default=15)
    max_temp = Column(Integer, default=25)
    pests = relationship("Pest",
                         secondary=plant_pest,
                         back_populates="plants")
    lights = relationship('light levels',
                          secondary=plant_light,
                          back_populates="plants")


    def __repr__(self):
        return f"<Plant({self.name})>"
```

```
class Pest(Base):
    __tablename__ = 'pest'
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)
    solution = Column(String, nullable=False)
    plants = relationship("Plant",
                          secondary=plant_pest,
                          order_by='Plant.name',
                          back_populates="pests")


    def __repr__(self):
        return f"<Pest({self.name})>"
```

```
plant_pest = Table('plant_pest',
                   Base.metadata,
                   Column('id', Integer, primary_key=True),
                   Column('plant_id', ForeignKey('plant.id')),
                   Column('pest_id', ForeignKey('pest.id')),
                   UniqueConstraint('pest_id', 'pest_id')
                   )
```

Database Criteria Fulfilment and Review

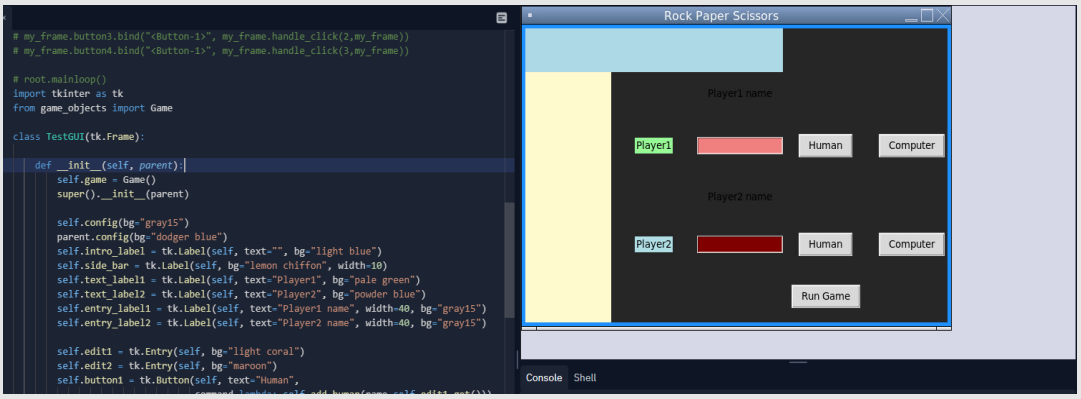
In order to properly review the Development Process, I need to compare each piece of development with the appropriate success criteria in order to ensure I am on track for the project and I don't deviate from the design or requirements.

R3	R5	D2	D3
Have a database system which contains multiple tables which the program accesses to display.	Characteristics shown include things like temperature, ease of maintenance, care instructions, size and what the plant looks like.	The database is made using SQL.	Use an image storing system using BLOBs (Binary Large Objects).
This is the fundamental of what this part of development is doing so whilst not finished yet, it certainly sets the program up the fulfil it.	By setting up the different fields of the tables I have set up what characteristics will be stored and now the rest of the program just has to receive them.	Simply by writing the code I am constructing the database in SQL.	By setting up the tables in SQL I am starting the pave the way to use BLOBs to store my images.

GUI

Refresher

I started my GUI development by using an old tkinter project designed to construct a rock paper scissors game in order to get back into using tkinter again since this had a very simple frame and grid setup. I realised quite quickly that this was ineffective and simply bad code but at the same time it worked as a good start point since it gave me a basis of understanding for the default layout of a tkinter program.

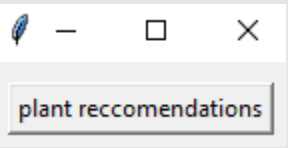


After this I started actually coding by creating a placeholder plant class which would act as my database inputs that the GUI would take. I used object oreinted programming because that's how it would be retrieved and allows me to easily refactor code by only changing parts at a time. At the same time I created a User Inputs class in the same file which determined how the inputs of the user would change in order to save them and edit what the reccomened plant list was off these values. As seen on the left these were simple classes with only basic methods such as setting all the attributes as well as an image attribute in the Plant class which will be touched on later.

PlantObjects.py

```
class Plant():
  def __init__(self,name,maxtemp,mintemp,brightness,size,ease,
  image):
    self.name=name
    self.maxtemp=maxtemp
    self.mintemp=mintemp
    self.brightness=brightness
    self.size=size
    self.ease=ease
    self.image=image


class UserInputs():
  def __init__(self,temperature,ease,size,brightness):
    self.temperature=temperature
    self.ease=ease
    self.size=size
    self.brightness=brightness

  def settemp(self,val):
    self.temperature=val
    print('set',val)
    print (self.temperature)

  def setease(self,val):
    self.ease=val
    print('set',val)

  def setsize(self,val):
    self.size=val
    print('set',val)

  def setbright(self,val):
    self.brightness=val
    print('set',val)

```

Main Menu



I began to work on the first phase of the GUI which was a simple menu for the user to click on, this needed to be simple so the user knows what they're doing.

```
class Menu(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.screenwidth = self.master.winfo_screenwidth()
        self.screenheight = self.master.winfo_screenheight()
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.plant_reccomendation_choice = tk.Button(self, text="plant reccomendations",
                                                     command=lambda: self.master.show_frame(self.master.frames[Page2]))
        self.plant_reccomendation_choice.grid(row=2, column=4, padx=10, pady=10)
```

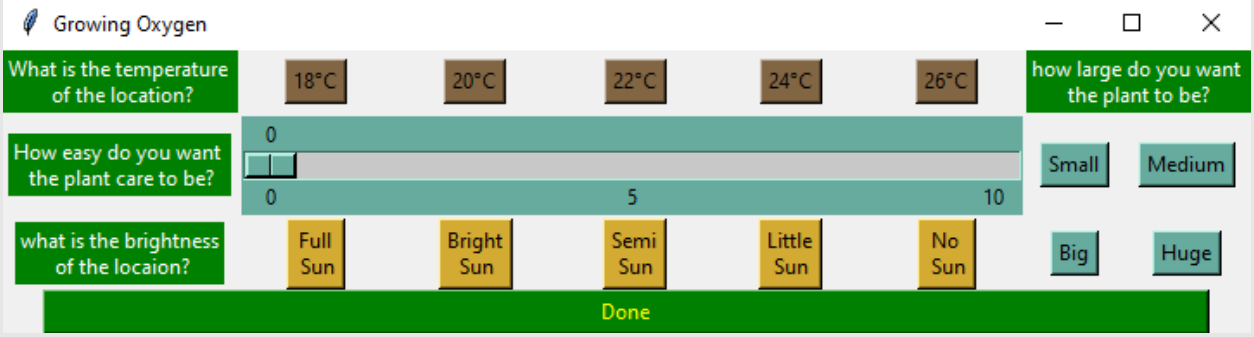
This was simple to do but also showcases many of the techniques used, the snapshot of this page is called a frame and the GUI essentially consists of many frames which are then switched through after being initialized in an overarching container class called PlantApp. The frames of the GUI are held in an attribute of the container class called *self.frames* and when the App is initiaized it creates all the frames in a dictonary structure calling them with their class name and then an initialization of the class passed with the self of the container class. This means then inherit the container as their master and can therefore control themselves from within themselves. This is important for the *show_frames* method of the container as shown on the right. This essentially turns the page through the dictonary by forgetting the objects in the previous frame and then packing the ones in the next one, leading the *show_frame* method to be called as *self.master.show_frame*(*frame to be shown*) this then allowed me to create buttons which easily forgot the current widgets without having to create a big forget function containing all the widgets. The use of this can be seen in the above image inside the highlighted square. Since all the first page consits of is a button leading to the next one it was easy, the second page was slightly more difficult.

```
    # Initialise frames to an empty dictionary
    self.frames = {}

    # Set up frames for each of the page classes
    pages = (Menu, UserInputsMenu, PlantContainer)
    for F in pages:
        frame = F(self)
        self.frames[F] = frame
    print(self.frames)
    self.show_frame(self.frames[Menu])

# Function to show the desired Page class, which is a subclass of tk.Frame
def show_frame(self, frame_to_show):
    self.forget_frames()
    frame_to_show.pack(expand=True, fill=tk.BOTH)

def forget_frames(self):
    widgets = self.winfo_children()
    # Forget all the frames
    for w in widgets:
        if w.winfo_class() == "Frame":
            w.pack_forget()
```

The class for this overarching container is called PlantApp and later on in the code will continue to fulfil its role as the master frame/container since the calculation algorithim will be called from within it. Initially the PlantApp class will be initialised in an object called app which then runs the inbuilt *self.mainloop()* method to launch the code.

```
app = PlantApp()
app.mainloop()
```

User Input Menu



The second phase of the GUI is incrediblly similar to the first in the sense thatit uses very similar techniques but just applies them more. The GUI displays 4 different parameters for the user to input into the system, brightness, size, ease and temperature, these parameters all deafault to None in case the user doesn't know/ want to input them. I started with designing the temperature buttons, this was quite simple since they were just buttons that called a function in the User Inputs class called settemp which assigns the ideal temperature. The named buttons such as the ones for size instead assign a number to the input, so small is 0 and huge is 3.

```
class UserInputsMenu(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.screenwidth = self.master.winfo_screenwidth()
        self.screenheight = self.master.winfo_screenheight()
        self.pack()
        self.plant_reccomendation()
        self.master.config(bg='green')

    def plant_reccomendation(self):
        #Label fot the buttons
        self.templabel = tk.Label(self, text="What is the temperature \n of the location?", bg="green", fg="white")
        self.templabel.grid(row=2, column=1)
        # buttons for setting temperature
        self.tempbutton1 = tk.Button(self, text="18°C", bg='#826644',
                                     command=lambda: david.settemp(18))
        #PLacing the button into the grid
        self.tempbutton1.grid(row=2, column=2, padx=1, pady=3)
```

All the buttons I have used have been using a function called lambda which restricts the definiton of a function to a single line instead of having a *def x():* essentially meaning that every time a button is pressed it calls the function this is used because otherwise each function would excecute simultaneously and the interface would not work. The excecution is due to the function being run when the widget is initilaisedso a lambda has to kind of buffer it through. This was something I had to understand with the next thing I implemented, the ease slider. The code for this can be seen below.

```
# slider for ease of care


self.easelabel = tk.Label(self, text="How easy do you want \n the plant care to be?", bg="green", fg="white")
self.easelabel.grid(row=3, column=1)
self.easeslider = tk.Scale(self, from_=0, to=10, length=450, tickinterval=5, orient='horizontal', bg="#67AB9F",
                           command=self.setease) #automatically passes self into the the function so doesn't need to be called
self.easeslider.grid(row=3, column=2, columnspan=5)
```

This was a challenge because it was not something I had tackled in class so when searching up a tutorial I was initally confused as to the way it used the lambda function to call a function without excecuting it. This means that instead of the function being excecuted with the slider value immediately, every time the slider changed the value of the slider was automatically passed into the lambda function leading it to set the ease each time which was what I wanted.

All the widgets in this frame are initialised in the frame by using the grid design instead of the pack method. The pack method essentially just adds on the widgets whereas the grid method can place a widget in a different place compared to the other widgets. The User Input Menu uses a 4*8 Grid as seen below.

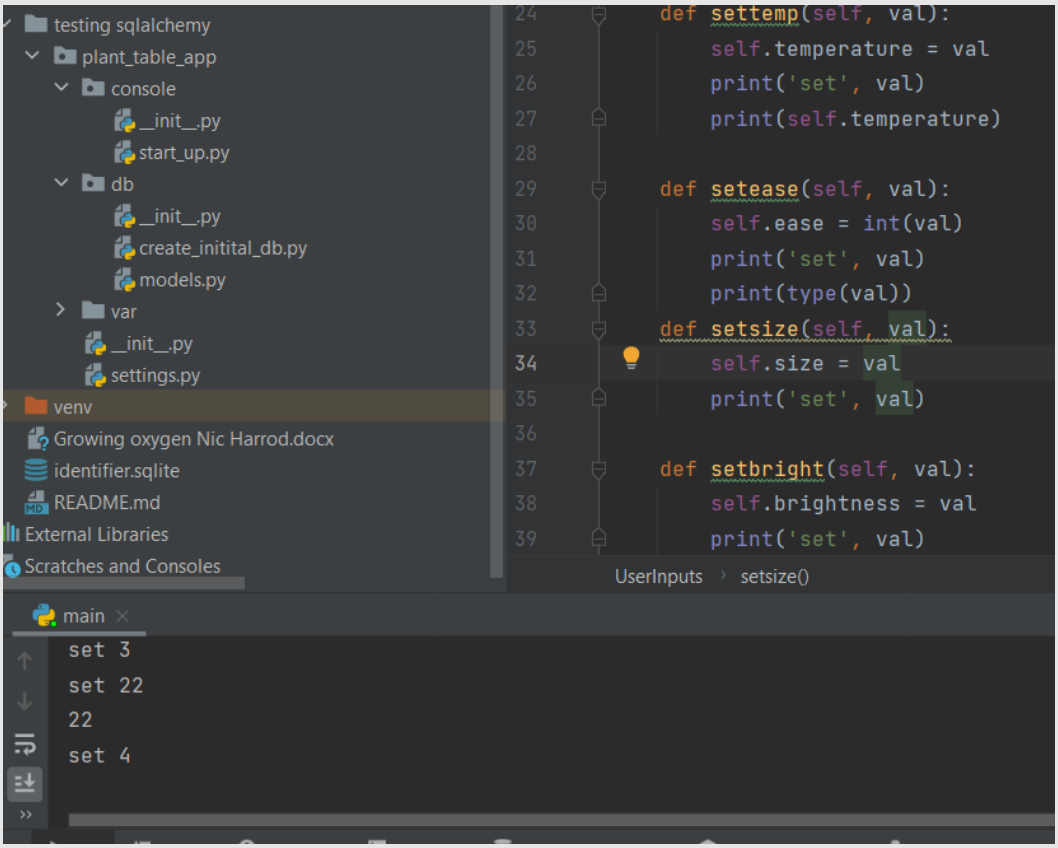
At this point I made a change of what attributes I would be showcasing for the plants, I decided to veto the idea of chemical vapor attributes since that is quite an unknown factor and is unlikely to come into consideration for any user choosing their plants.

Temp Label	18	20	22		24	26	Size Label	
Slider Label		Ease Slider					Small	Medium
Brightness Label	Full Sun	Bright Sun	Semi Sun		Little Sun	No Sun	Big	Huge
	Done Button							

The grid method is far easier to use for something like this because I want a lot of uniform widgets that can be grouped together, for example all the temperature buttons are on row 1 (or row 2 for the code) , making it far easier to visualise instead of the pack method which you'd need to estimate how it looks from the size of your widgets.

```
self.templabel = tk.Label(self, text="What is the temperature \n
self.templabel.grid(row=2, column=1)
self.tempbutton1 = tk.Button(self, text="18°C", bg='#826644',
                             command=lambda: david.settemp(18))
self.tempbutton1.grid(row=2, column=2, padx=1, pady=3)
self.tempbutton2 = tk.Button(self, text="20°C", bg='#826644',
                             command=lambda: david.settemp(20))
self.tempbutton2.grid(row=2, column=3, padx=1, pady=3)
self.tempbutton3 = tk.Button(self, text="22°C", bg='#826644',
                             command=lambda: david.settemp(22))
self.tempbutton3.grid(row=2, column=4, padx=1, pady=3)
self.tempbutton4 = tk.Button(self, text="24°C", bg='#826644',
                             command=lambda: david.settemp(24))
self.tempbutton4.grid(row=2, column=5, padx=1, pady=3)
self.tempbutton5 = tk.Button(self, text="26°C", bg='#826644',
                             command=lambda: david.settemp(26))
self.tempbutton5.grid(row=2, column=6, padx=1, pady=3)
# slider for ease of care
```

The naming convention of the temperature buttons is simple to keep the code ledgible, this was kept along the rest of this frame, with sizebutton1 and brightbutton1. This means that the code is far easier to understand from a glance because it can be seen what the button is for and also how many there are, making the visualisation of the UI far easier.



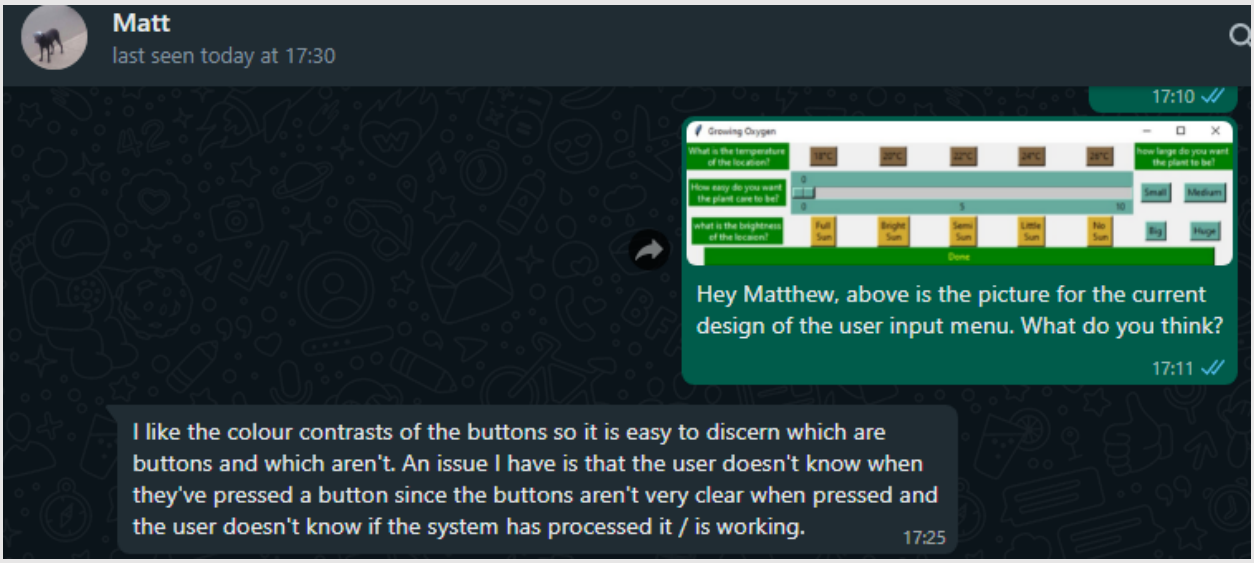
The set[var] functions can be seen to the left, these were important in order to provide an object for the calculations to accept, the print functions within them exist to prove that they work before I actually saw the result as can also be seen.

After I had finished with the main part of the user input menu code I had to start work on the done button for it, initially all I had it do was change the frame and leave the calculations for later so it was a simple change frame, also in order to see if the attributes had changed I printed out the user inputs class through the David temporary object which is initialized at the start with default attributes of None.

```
def attributesdone(self):

    print('done', david.temperature, david.ease, david.size, david.brightness)
    self.master.show_frame(self.master.frames[PlantContainer])
```

User Input Menu Review

Since the user input menu was the part of the GUI design that I had fleshedout the least, I wanted to submit this design to my stakeholders first. I submitted it to Matthew since he was more versed on process rather than aesthetic.

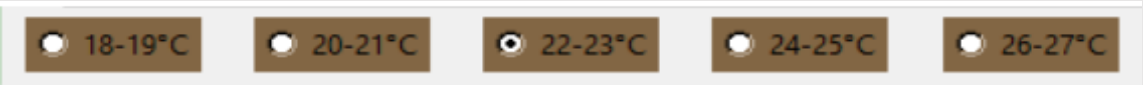


Matthew said that he didn't like that the user wasn't sure if a button had been pressed. The solution to this is to use radio buttons. Radio buttons are similar to normal buttons except you can only select one at a time and it marks them as selected. Tkinter contains a radio button class that is very similar to the normal button object. This is advantageous since it means that I didn't have to change much of the layout of the parameters. The new parameters passed into it were a varaiable to group the buttons together initialsed as an IntVar so it only exists within the frame, and also a value to separate the buttons. Below are the before and after of the temperature button setting up, with the new pieces of code marked in red.

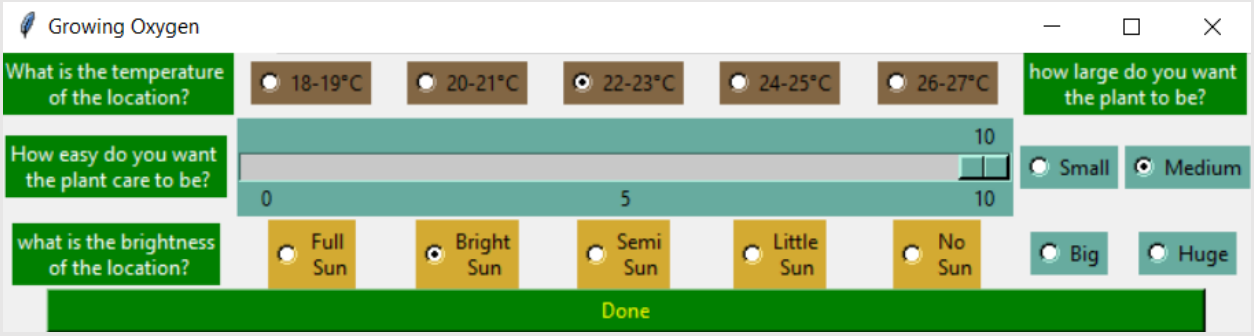
```
def plant_reccomendation(self):
    #Label fot the buttons
    self.templabel = tk.Label(self, text="What is the temperature \n
    self.templabel.grid(row=2, column=1)
    # buttons for setting temperature
    self.tempbutton1 = tk.Button(self, text="18°C", bg='#826644',
                                 command=lambda: david.settemp(18))
    #PLacing the button into the grid
    self.tempbutton1.grid(row=2, column=2, padx=1, pady=3)
```

```
def plant_reccomendation(self):
    var1=tk.IntVar(self)
    # Label fot the buttons
    self.templabel = tk.Label(self, text="What is the temperature \n of the locatio
    self.templabel.grid(row=2, column=1)
    # buttons for setting temperature
    self.tempbutton1 = tk.Radiobutton(self, text="18-19°C", bg='#826644',value=1,
                                      variable=var1,command=lambda: GOxygen.settemp(18))
    # PLacing the button into the grid
    self.tempbutton1.grid(row=2, column=2, padx=1, pady=3)
```

When running the program the difference was clear as seen below with the 22-23 temperature button selected, the user can celarly see what temperature is currently chosen.



After this I went to start turning all the buttons in the user input menu to radiobuttons. Below is the final product for the user input menu with radio buttons.



Below is the table for the Criteria that the User Input Menu Fulfils.

R1	R2	D5
Create a program that recommends plants based on attributes of the user's room.	Have a user interface which takes inputs from the user and then does the calculations for the recommendation.	Whilst needing to look good the GUI also must be simple enough to understand in order to be appropriately efficient.
By taking the attributes of the user, the user input menu fulfils this.	The user input menu currently takes the user inputs and stores them.	The inclusion of radio buttons and explanation labels makes the GUI far easier to understand than not having it.

Plant Boxes

Now it was time to move onto the most technically complex part of the GUI, the plant list display. Initially I created a Plant List temporary small list which just contained Plant Objects because the idea for this was to display the plants in a container.

```
Plantlist = [
    P('Devils Ivy', 24, 18, 3, 2, 9, 'devils ivy.PNG'),
    P('Peace Lily', 26, 20, 4, 1, 7, 'Peace Lily.png'),
    P('Snake Plant', 24, 18, 4, 3, 4, 'snake plant.png')
```

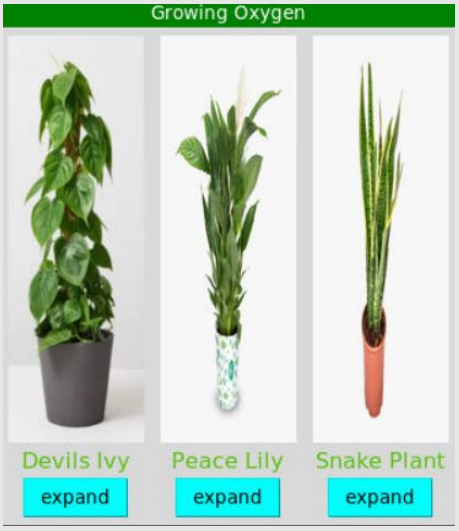
```
class PlantContainer(tk.Frame):

 def __init__(self, master):
   super().__init__(master)
   self.master = master

   self.frames = [ PlantBox(self, plant) for plant in Plantlist]
   for pb in self.frames:
     pb.pack(side=tk.LEFT, padx=5, pady=5)
```

On the left is the initial layout for the Plant Boxes, what this did was create a container object which could be shown as a frame and within the class is a for loop that iterates through the Plant List and creates a Plant Box object for each value and after packs it into the container. I am using the pack instead of Grid method for the container because all I need to do is put them next to each other, which pack does, the padding on the pack method just makes the GUI look nicer in this context so nothing looks to cramped on the user's screen.

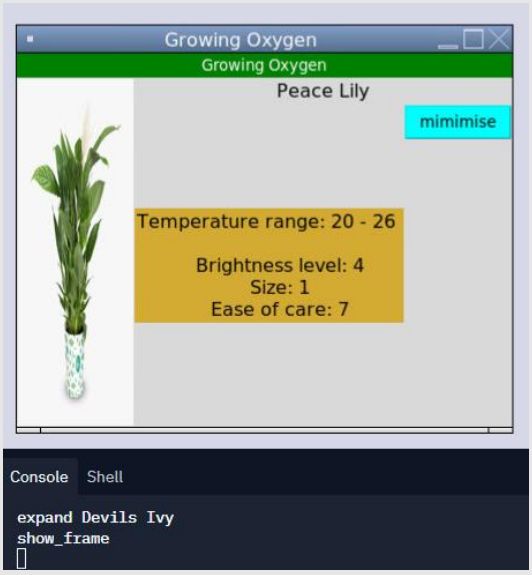
```
class PlantBox(tk.Frame):


 def __init__(self,master, plant):
   super().__init__(master)
   self.master = master
   self.name=plant.name
   with Image.open(plant.image) as im:
     im_resized = im.resize((100, 300))
     im_resized.show()
     self.image=ImageTk.PhotoImage(im_resized)
     #self.image=tk.PhotoImage(file=Plantlist[val].image)
   self.imagelabel=tk.Label(self,image=self.image)
   self.imagelabel.pack()
   self.boxtitle=tk.Label(self,text=self.name,fg='yellow',
   font='bold')
   self.boxtitle.pack()
   self.expandbutton=tk.Button(self,text='expand',bg='cyan',
   command=lambda:self.expand(plant))
   self.expandbutton.pack()
 def expand(self,pl):

   print(f'expand {self.name}')
   self.master.master.clear_frame()
   self.master.master.__init__(pl)

   self.master.master.show_frame(PlantExpand)
```

Left is the first iteration of code for the Plant Box, which takes a plant object as a parameter. Notably there is an image opening part which is new for the code, what this does is convert a PNG into a Tk object by overlaying onto a label of set size, this means that the image stretches to fit the size of the Label, meaning all the plant boxes are the same size, as indicated by the parameters given into *im_resize()*. The init (constructor) method of the Plant Box also creates a label according to the name of the class passed into it. The expand button initialized after turned out to be one of the most difficult parts of the code so far.

Left is the picture of the plant container with the plant boxes displayed. You can see how well the padding works since they aren't too bunched together but at the same time the image is quite squashed so I might end up copping them somehow or just making them all the same size in the files, but only if I have the time.

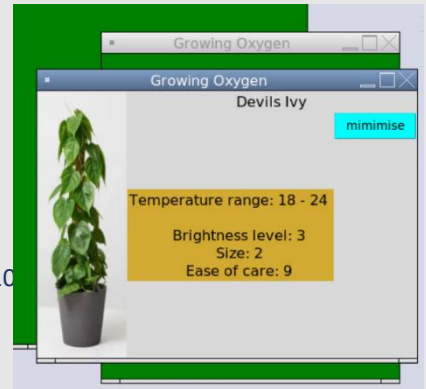


The issue with the expand button is that it was seeming to change the frame into the expanded information about the plant that was clicked on but so far throughout the GUI I had used the *show_frame()* method to display a new item. This resulted in the new frame only ever showing the plant irrespective of what plant had been expanded, for this I had passed the peace lily into it so even if Devil's Ivy was expanded it would show Peace Lily as seen above. In the expand function I also had it print the plant name, so I knew which plant was being expanded and to also check that the thing that was being expanded was changing. This was because when the *show_frame()* method was initially created it made the *frames* dictionary which packed all the frames at the start. Since the PlantExpand class was part of *self.frames*, it meant that the plant that the class took would not be able to change without reinitializing the master class. Because of this I had to initialize the class to have the Peace Lily as the chosen plant.




```
class PlantExpand(tk.Frame):

  def __init__(self,master=None):

    super().__init__(master)
    p=self.master.expandedChoice
    print ('hiiiiiiiiii',p.name)
    self.master = master
    self.name=p.name
    with Image.open(p.image) as im:
      im_resized = im.resize((100, 300))
      im_resized.show()
      self.image=ImageTk.PhotoImage(im_resized)
    self.imagelabel=tk.Label(self,image=self.image)
    self.imagelabel.pack(side=tk.LEFT)
    self.textbox=tk.Label(self,bg='#D3AA32',text=
    f'''Temperature range: {p.mintemp} - {p.maxtemp} \n
    Brightness level: {p.brightness}
    Size: {p.size}
    Ease of care: {p.ease}''',font=(44))
    self.textboxtitle=tk.Label(self,text=p.name,font='bold')
    self.textboxtitle.pack(side=tk.TOP)
    self.textbox.pack(side=tk.LEFT)
    self.minimisebutton=tk.Button(self,text='mimimise',bg='cyan',
    command=lambda:self.master.show_frame(PlantContainer))
    self.minimisebutton.pack(side=tk.TOP)
```

Other than the actual initializing of the object, the PlantExpand class is quite self-explanatory, it has the same image defining parts as the Plant Box and other than that it just has a list of attributes and will eventually have a small paragraph as well which shows how to

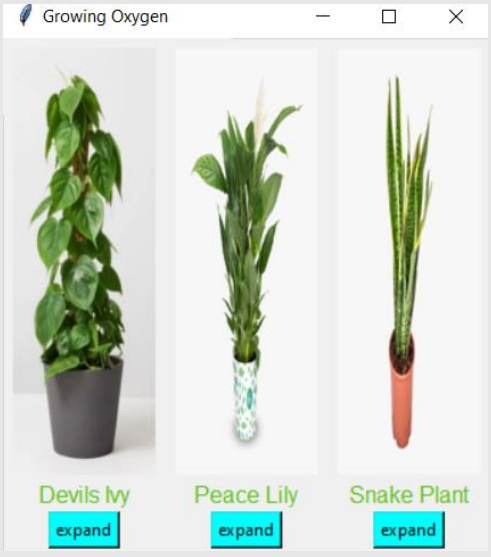


manage care along with the pest information and other details on the plant not just given from the User Inputs. At the end of the class is the button for switching back to the plant selection which very simply reverts back to the Plant Container in the original frame. I initially thought that the best way of fixing it was creating a global variable called *recommended* plant which when I showed the Expand frame, but this didn't work since I had not yet realized the issue in packing.

After this I tried to fix the issue by reinitializing the entire PlantExpand class by just running the constructor method again in PlantBox, the issue with this is that it created a new container and a whole new window whilst keeping the old one, this can be seen right. The fact that the Devil's Ivy plant information is being shown means that at least the code works for changing the expanded plant. After I found this I started trying to just change the *show_frame()* method in order for it to delete the previous container. I quickly found that in the end reinitializing was a duct tape solution and there was no way to make this work the way I wanted it to.

```
class PlantBox(tk.Frame):
    def __init__(self, master, plant):
        super().__init__(master)
        self.master = master
        self.frames = {BoxPhase1: BoxPhase1(self, plant), BoxPhase2: BoxPhase2(self, plant)}
        self.show_frame(self.frames[BoxPhase1])

    def show_frame(self, frame_to_show):
        self.forget_frames()
        frame_to_show.pack(expand=True, fill=tk.BOTH)

    def forget_frames(self):
        widgets = self.winfo_children()
        # Forget all the frames
        for w in widgets:
            if w.winfo_class() == "Frame":
                w.pack_forget()
```

My final solution to this issue was to make the Plant Box class into a similar object to PlantApp by giving it it's own frames dictionary and *show_frame()* function, this means that each plant box acted as a container which then treated either the first or second phase as separate frames which could easily be switched between. The show and forget frames are the same as before but the *self.frames* dictionary is different because in this version it sets up the two 'phases' of the plant box, the first phase is the 'unexpanded' version whereas the second is the 'expanded' version, this makes the expand button far easier to revert back because all it does is show another frame.



An advantage of this solution is that it shows the expanded version as well as keeping the previous frames so if the user wants they can compare the statisitcs of other plants to find out the one they really want.

After this I also did some maintenance to clean up the BoxPhase frames, while these were very similar to the original expand I wanted to touch up how they looked and explore them more.

On the left is the first frame of the plant box, this is not particularly different but I used better formatting and cleaned up the layout so it fitted with PEP-8.

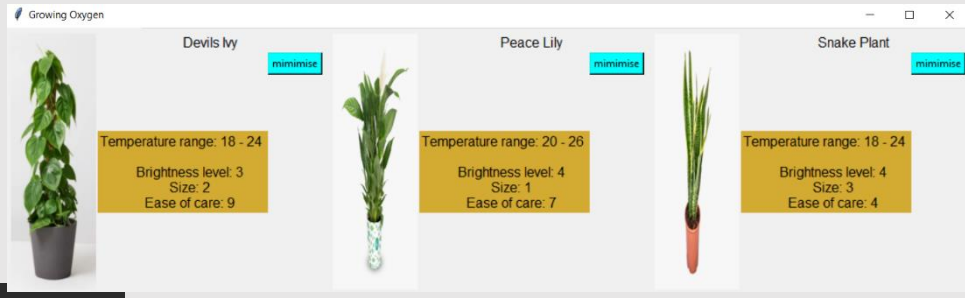
This time the expand function works similar to the other frames in the code, it accesses the methods of its parent container and executes the *show frame()*

method but instead of accessing the plant app frames it goes just to the plant container class for them, this is changed because it only goes one layer up. The object it's initialized in acts as the 'master' so that the frame can access its methods. Because of this *self.master.[function]* will access that frame's container's function and so on. In one of my solutions for the plant box issue, I used

```
class BoxPhase1(tk.Frame):

    def __init__(self, master, plant):
        super().__init__(master)
        self.master = master
        self.name = plant.name

        with Image.open(plant.image) as im:
            im_resized = im.resize((100, 300))
            self.image = ImageTk.PhotoImage(im_resized)
        self.imagelabel = tk.Label(self, image=self.image)
        self.imagelabel.pack()
        self.boxtitle = tk.Label(self, text=self.name, fg='#5BC014', font='bold')
        self.boxtitle.pack()
        self.expandbutton = tk.Button(self, text='expand', bg='cyan', command=lambda: self.expand(plant))
        self.expandbutton.pack()

    def expand(self, pl):
        print(f'expand {self.name}')
        self.master.forget_frames()
        self.master.show_frame(self.master.frames[BoxPhase2])
```

self.master.master.show_frame(PlantExpand) this was just sloppy code and so with this plant box frame I have solved the problem and removed any double masters from my code as shown by the red box.

Below is the code for the second frame of this, initially the PlantExpand frame. I have changed this in a few ways. First is *self.p = plant*, this is different from the previous one since it had just been *self.name* but this time it takes the whole plant class and stores it as an attribute, this saves it and isn't temporary like it would be if just saved as an input. Below is also the GUI for when all the plant boxes are expanded.




```
class BoxPhase2(tk.Frame):

    def __init__(self, master, plant):
        super().__init__(master)
        self.p = plant

        self.master = master
        self.name = self.p.name
        with Image.open(self.p.image) as im:
            im_resized = im.resize((100, 300))
            self.image = ImageTk.PhotoImage(im_resized)
        self.imagelabel = tk.Label(self, image=self.image)
        self.imagelabel.pack(side=tk.LEFT)
        self.textbox = tk.Label(self, bg='#D3AA32', text=
        f'''Temperature range: {self.p.mintemp} - {self.p.maxtemp} \n
    Brightness level: {self.p.brightness}
    Size: {self.p.size}
    Ease of care: {self.p.ease}''', font=(44))
        self.textboxtitle = tk.Label(self, text=self.p.name, font='bold')
        self.textboxtitle.pack(side=tk.TOP)
        self.textbox.pack(side=tk.LEFT)
        self.minimisebutton = tk.Button(self, text='mimimise', bg='cyan',
                                        command=lambda: self.contract())
        self.minimisebutton.pack(side=tk.TOP)
    def contract(self):
        self.master.forget_frames()
        self.master.show_frame(self.master.frames[BoxPhase1])
```

Currently the list provided in the plant boxes is the same each time because it's retrieved from the plant list array, in order to fix this I would have to design calculations so I actually had something to do with the buttons in the User Input menu.

Plant Boxes Criteria Fulfilment and Review

Below is the table for which success criteria the Plant Boxes fulfil.

R4	R5
Display the characteristics of the recommended plant and what they look like.	Characteristics shown include things like temperature, ease of maintenance, care instructions, size and what the plant looks like.
This is the underlying reason for the Plant Boxes.	Currently being displayed are all of these.

Algorithm

Implementing

```
        self.desirability_score = 0

    def scorechange(self, val):
        self.desirability_score += val
```

To start doing this I had to change the Plant Class to include a numerical value known as *desirability_score* Since I wasn't currently using a database. In order to access this score, I had to change where the Plant List was stored by assigning it as an attribute inside the PlantSort class. This would aslo give me a concrete place this list was stored because previously it could have been anywhere such as when it was just shoe-horned in at the end of the code for the GUI.

```
from PlantObjects import Plant as P

from PlantObjects import UserInputs


class PlantSort:
    def __init__(self, userinputs):
        self.PlantList = [
            P('Devils Ivy', 24, 18, 3, 2, 9, 'devils ivy.PNG'),
            P('Peace Lily', 26, 20, 4, 1, 7, 'Peace Lily.png'),
            P('Snake Plant', 24, 18, 4, 3, 4, 'snake plant.png')]
        self.UI = userinputs
```

Next the first task was to start working on the calculations, an advantage of the UserInputs Menu was the fact that all the inputs were numerical and therefore easy to manipulate. I went via the rough guideline that if a plant had a value that was the exact 'desired' value then it would subsequent oin 5 points. The method for scoring the plans was simply using a for loop to go through them and then accessing the chosen attribute. I chose the brightness attribute to start with since that was the simplest, the score for the adjacent values in this case was simple because it was a simple range, if bang on the value it was +5 and if 1 away +2, 2 away and +1, anything else would be 0 because if there's a plant that needs high sun and the room is very dark then theres no way it would survive.

```
def score_bright(self):
    if self.UI.brightness:
        for i in self.PlantList:
            if self.UI.brightness == i.brightness:
                i.scorechange(5)
            elif self.UI.brightness == i.brightness + 1 or self.UI.brightness == i.brightness - 1:
                i.scorechange(2)
            elif self.UI.brightness == i.brightness + 2 or self.UI.brightness == i.brightness - 2:
                i.scorechange(1)
```

```
def score_size(self):
    if self.UI.size:
        for i in self.PlantList:
            if self.UI.size == i.size:
                i.scorechange(5)
            elif self.UI.size == i.size + 1 or self.UI.size == i.size - 1:
                i.scorechange(3)
            elif self.UI.size == i.size + 2 or self.UI.size == i.size - 2:
                i.scorechange(1)
```

The strategy for designing these calculations was the same for some of the other ones, such as temperature as seen above. The calculations varied again with the ease calculation since this had a far larger range of values. Theresefore I had to create a formula. If a plant's ease level was the same as the

desired value then I had to give it 5 as I said earlier, therefore the base of the formula would start from 5. Then as the difference between the values increase, the score for the plant would decrease, so as the modular value for one minus the other increased, the resultant value must go down. Therefore I was able to construct a formula looking like this: $x = 5 - |P_v - D_v|$ where x is the assigned score, P_v is the plants value for ease and D_v is the user input for ease. The advantage fo this is if something if drastically far away from the deisred value of ease then it isn't counted.

```
def score_ease(self):
    if self.UI.ease:
        for i in self.PlantList:
            value = 5 - abs(self.UI.ease - i.ease)
            # if something is greater than 4 away from the desired ease, it will not be counted
            if value <= 0:
                pass
            else:
                i.scorechange(value)
```

The hardest attribute to score was the temperature attribute, this is because all plants had a maximum and minimum temperature value. I started by simply making the score change by 5 if it was within the range of max and min, since then it had reached the desired value. Then I had to calculate how the score would change if it was outside the range, since the temperature guidelines for most plants are rough guides anyway. I did this by creating a function called *find_temp_dist()* this stored a list of all possible temperatures, an advantage of this data is that it's discrete so can be easily sorted through. The function takes the maximum and minimum temperature values for the plant as well as the user temperature value, then it finds how far the user temperature is to it's closest upper or lower bound, if the temperature value is higher than the max, it finds the distance between the higher user input and the maximum temperature (user-max) ; if it is lower than the min it finds the distance between the lower user input and the minimum temperature (min-user).

```
def score_temp(self):
    if self.UI.temperature:

        for i in self.PlantList:
        # seeing if it is within the temperature range
            if i.mintemp <= self.UI.temperature <= i.maxtemp:
                i.scorechange(5)
            else:
                i.scorechange(self.find_temp_dist(self.UI.temperature, i.maxtemp, i.mintemp))

def find_temp_dist(self, tempval, max, min):
    # finding distance from ideal value of temperature in order to appropriately score the plant
    a = [16, 18, 20, 22, 24, 26]
    if tempval > max:
        return a.index(tempval) - a.index(max)
    if tempval < min:
        return a.index(min) - a.index(tempval)
```

```
def calculate(self):
    self.score_bright()
    self.score_temp()
    self.score_size()
    self.score_ease()
    self.PlantList.sort(key=lambda plant: plant.desirability_score, reverse=True)

def best(self):
    self.calculate()
    return self.PlantList[0]
```

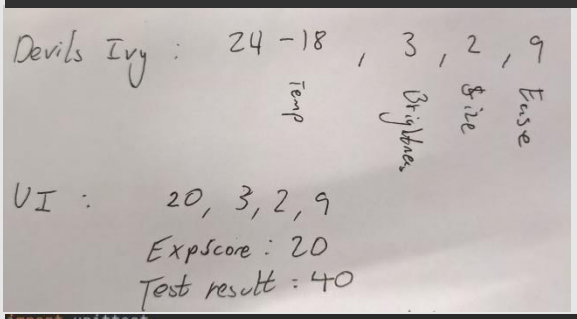
Instead of grabbing the actual values of temperature, it retireves their position in the temperature list using *.index()* this was so that the difference wasn't too big and no further calcualtion had to be done after the difference calculation.

After this I put it together in a function called *self.calculate()* which very simply ran all the scoring functions and then sorted the list according to the score, this used the *.sort()* procedure that's built into lists. Since this was a list of objects I instead had to change it by using a lambda, this meant that for every object in the list (I named these objects plant) it would get the objects desirability score.

Unit Tests

Now I had to test that the calculations I had implemented worked. This was done using a python module called unittests, where executable methods were defined with placeholder values and then sent off in order to see if an algorithm works. I started this by verifying what I wanted to test by creating a *self.best()* method inside of PlantSort which would calculate the list and then spit out the first value, i.e. the best plant. I initially wanted to test the calculation for Devil's Ivy so I calculated what the expected value should be based off if the user inputs were the exact values that lined up with the attributes of the plant.

```
def best(self):

    self.calculate()
    return self.PlantList[0]
```



This means that if I ran the *.best()* method on a plant sort with these user inputs, it would return devil's ivy.

Below is the setup for the first unit test I did which was just ensuring that it reurns a good result and then seeing if the calculated score was the same as the returned one, as you can see on the right of the code, when the test ran it failed and this was due to the fact that the calcualted score was different to the expected score, 20 was not equal to 40 but at the same time the previous test passed so devils ivy was being returned which means the issue was definietly in calculations.py. After looking around the code I found that the issue was with a small test variable I did with the calculations while initally making them as seen below. This was in tandem with the various variable names I gave whilst testing individual parts of the code, I named them using the naming convention of common first names because it made them easier to view as items other than normal temporary names such as 'a' or 'var'. It turns out that when I imported the calculations module it affected the desireablity score for that instance of the plant object by running james.calculate so all I had to do was remove it and then the problem would be solved.

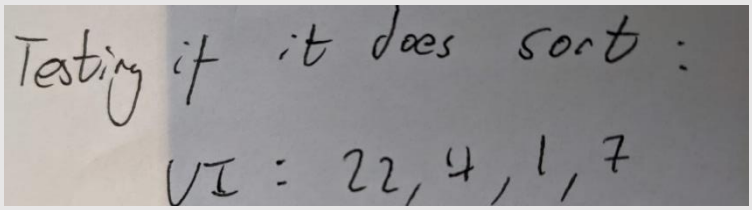
```
import unittest
import calculations as c
from GUI_Code import PlantObjects as po


class CalculationsTest(unittest.TestCase):
    def setUp(self):
        self.TempPlant = c.PlantSort(po.UserInputs(20, 3, 2, 9)).best()

    def test_devils(self):
        #testing that it returns a result
        self.assertEqual(self.TempPlant.name, "Devils Ivy")
    def test_devilscore(self):
        #testing that the result is the correct score
        self.assertEqual(self.TempPlant.desirability_score,20)
```

```
Ran 2 tests in 0.013s

FAILED (failures=1)


20 != 40

Expected :40
Actual   :20
```

```
daniel = UserInputs(20, 3, 2, 9)
james = PlantSort(daniel)
james.calculate()
print([i for i in james.PlantList])
```

```
Ran 2 tests in 0.004s

OK
```

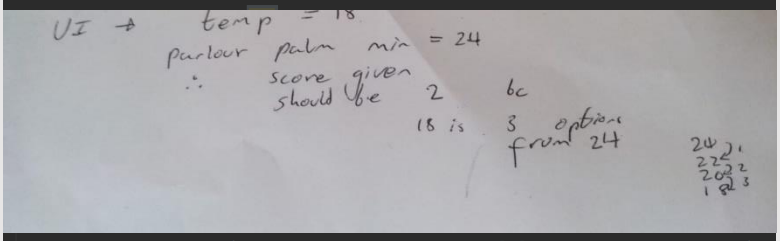
After remvoing this test case from my file I was able to succeed in my tests as shown to the right. Next was to test other things, like if the list sorted correctly, this was done by creating another TempPlant variable and assigning it to the value that would return the parlour palm since this wasn't first in the list, if best() didn't return this value then it meant that it was still just taking it from the first item in the list, devil's ivy.




```
        self.TempPlant2 = c.PlantSort(po.UserInputs(22, 4, 1, 7)).best()

    def test_devils(self):...

    def test_peace(self):
        self.assertEqual(self.TempPlant2.name, "Peace Lily")
```

```
peace
Tests passed: 1 of 1 test – 2 ms

Ran 1 test in 0.002s

OK
```

```
    self.TempPlant3=c.PlantSort(po.UserInputs(18,None,None,None))
```




```
    self.assertEqual(self.TempPlant3.PlantList[1].desirability_score,2)
```

Since the test passed this means that the best method returned parlour palm. After this I wanted to test the way score was distributed for sorting and ranking the plants, the most likely to fail version of the scoring was the temperature score since that had a complex way of allocating the score, if something was within it then it should get 5 score and if it was outside it should receive score inversely proportional to how far away it was, if the distance was greater then the score should be lower. In order to do this test I created a new temp plant variable which was simply the calss of a plant sort after being sent the UI values of 18 degrees and None for all the others, this means that a plants score would only be calculated by the temperature. For the parlour palm with these values, the score should be calculated to be 2 after this. Since the list wouldn't be sorted I was able to get the score by just accessing the 2nd item in the list or [0].

When I ran this test it failed because the score it spat out was 3, meaning that the further something was from the ideal the lower score it would get. In order to fix this I had another look at the temperature distance calculation and it turned out that it was just calculating the size of the distance, not the score that should be attributed, to change this I subtracted the length from

```
Ran 1 test in 0.007s

FAILED (failures=1)

2 != 3
```

5 in order to get a good measure of score, so if the distance was 1 then the score would be 4 and if it was 3 then the score would be 2.

```
if tempval > max:
    return a.index(tempval) - a.index(max)
```

```
if tempval > max:
    return 5 -(a.index(tempval) - a.index(max))
```

Then when I ran the test it produced this on the right, finalising this I had finished the calculations part of the code and now the job was to implement it in the GUI.

```
Ran 1 test in 0.005s

OK
```

Algorithm Criteria Fulfilment and Review

Below is the table for the criteria that the algorithm so far completes.

R2	D4
Have a user interface which takes inputs from the user and then does the calculations for the recommendation.	Have a calculated ranking that is based off the user inputs.
This is the calculations part of this criteria	The fact that the Calculations returns the whole list of plants and not just the best one ensures this.

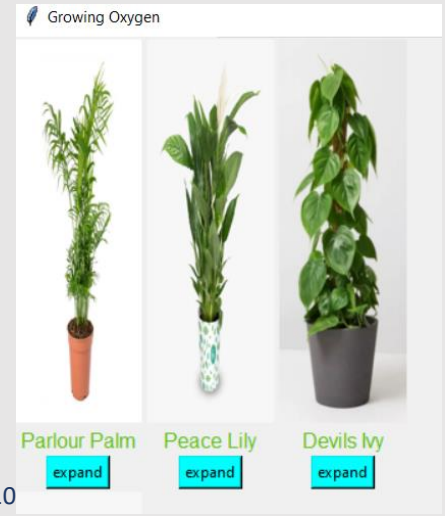
GUI Implementation

Now what I had to do was implement the calculation into the GUI, in theory this was easy as all I would do is change the way I had set the GUI up so that when the user clicked the 'done' button on the user inputs menu then it would run the calculations and give an ordered plant list for the plant container to put into the plant boxes. This soon changed as I ran into the issue of nothing changing since the plant container frame that was shown was the plant list that had been first set up when the list was and frames dictionary was packed, the done button didn't re-pack the frame. In order to fix this it meant I needed the done button to change an attribute of its master class, PlantApp I did this by creating a function in it called *sort_PlantList()* which created a plant sort class using the user inputs from the David object so as the attributes of the David object changed, it would affect how it was then calculated since the David object was an instance of the User Inputs class. What accessing the self. Frames dictionary did is by altering the data stored at the PlantContainer index and repacking all the items but this time with the calculation variable so that the plant container would be able to access the new plant list. This was accessed in the done button by putting it in the *.attributesdone()* method which would call the plant app function through *self.master*. Now I needed to edit the plant container so it would be viable for the new calculation list. The biggest issue was that it needed to take an initial list but then know after it was packed that it needed to take the new plant list. In order to do this I added a new attribute into the contructor which was a flag that defaulted to None in order for it to work before being sorted but then when *sort_PlantList* passes the calculation class into the plant container it would need to change. So I branched the creation plant boxes depending if the PlantSort object was present,

```
class PlantContainer(tk.Frame):

    def __init__(self, master, Plist=None):
        super().__init__(master)
        self.master = master
        if not Plist:
            self.frames = [PlantBox(self, plant) for plant in Plantlist]
            for pb in self.frames:
                pb.pack(side=tk.LEFT, padx=5, pady=5)
        else:
            count=0
            self.frames = [PlantBox(self, plant) for plant in Plist.PlantList]
            for pb in self.frames:
```

```
def sort_PlantList(self):
    # sorting the list of Plants
    calculation = PS(david)
    calculation.calculate()
    self.frames[PlantContainer] = PlantContainer(self, calculation)
```

```
    self.DoneButton = tk.Button(self, text='Done', bg='green', fg='yellow', width=95
                                command=lambda: self.attributesdone())
    self.DoneButton.grid(row=5, column=1, columnspan=9)

def attributesdone(self):
    self.master.sort_PlantList()
    print('done', david.temperature, david.ease, david.size, david.brightness)
    self.master.show_frame(self.master.frames[PlantContainer])
```



essentialy acting as its own flag. When running this I was surprised that it worked first time which was a first for this project. As seen right, the plantbox order has changed from what the initial plantlist is, meaning that the list had been sorted according to the user inputs.

Gui Implementation Criteria Fulfilment and Review

Below is the table for the criteria that the implementation of the algorithm in the GUI completes.

R1	R2
Create a program that recommends plants based on attributes of the user's room.	Have a user interface which takes inputs from the user and then does the calculations for the recommendation.
This is implementing the recommendations into the GUI; therefore, the program is now succinctly recommending plants.	By integrating the algorithm and GUI I am doing this since the calculations take the data from the GUI as a parameter.

Databases (Continued)

After finalizing all the front and back-end parts of the code, it was time for me to link it to my database, unfortunately I had run out of time to go back into playing around with SQLAlchemy , therefore what I needed to do was find another solution.

The solution I found was to continue using the class and array structure I had already started using for my placeholder testing. I had to flesh out the fields and form connections from one array to another. An advantage of this is that it keeps the project wholly in python meaning that it would make the project easier to run and download in terms of modules. Another advantage of the class system is that it works just as well as SQL for small databases like mine is going to be, especially since I don't intend to do any complex queries. I started this by creating a Database.py file where I would store all this information.

Database.py

This file just contained the plant list as seen below except I needed to refine it by adding more plants to it, I started this by adding 3 more plants in order to appropriately see how it would look with more items. I then added more attributes to the plants by constructing a pest list and a care information small message. This was in order for the user to get more information from the plant they were

```
from GUI_Code.PlantObjects import Plant as P
#name, maxtemp, mintemp, brightness, size, ease, careinfo, pests, image


PlantList = [
            P('Devils Ivy', 24, 18, 3, 2, 9,'Water once a week and mist',[0,1], r'C:\Users\nicp
            P('Peace Lily', 26, 24, 4, 1, 7,'Water once a week and prune flowers',[2,1], r'C:\U
            P('Snake Plant', 24, 18, 4, 3, 4, 'Water every other week and dust',[0],r'C:\Users\
            P('Parlour Palm', 20, 18, 2, 3, 7, 'Water every week and repot once a year',[0],r'C
            P('Caladium',22,18,5,2,3,'Water barely and mist regurlarly',[1],r'C:\Users\nicpa\Py
            P('Orchid',24,22,5,2,1,'water never and occasionally repot',[2],r'C:\Users\nicpa\Py
]
```

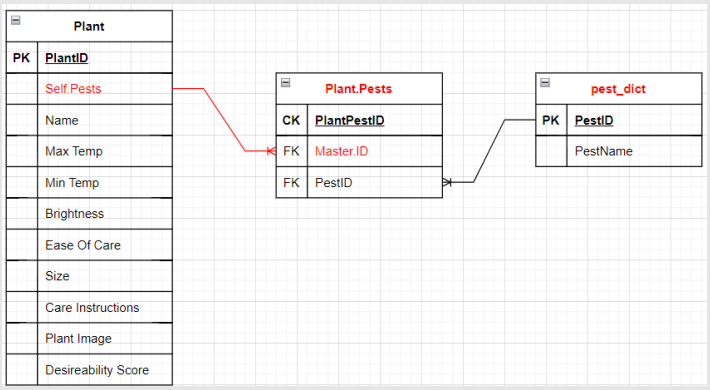
recommended than what they put into the user inputs menu.

Since I still wanted some database relationship aspects, I constructed a pests class in plant objects that contained a pest dictionary that the plant object would look at and get information from. This was done in the plant objects code since the pests had to be retrieved for the construction on the instances of the plant class. This class used the *pest_dict* dictionary as a table for the pests, with the pest ID as the index for the pest which then used the Pests

```
pest_dict = {0: 'Aphids ', 1: 'Leaf Miners ', 2: 'GreenFlies '}


class Pests:
    def __init__(self, master, pestlist):
        self.parent = master
        self.list = [pest_dict[i] for i in pestlist]

    def __repr__(self):
        return f'{self.parent.name} pests : {"".join(self.list)}'
```

```
self.pests = Pests(self, pests)
```

class to insert it into *self.pests* for the Plant Class. E.g., for the Devil's Ivy object, when it is initialized the pest indexes of [0,1] are passed into *self.pests* which initializes a Pests class which then creates a *self.list* of the indexed pests from the *pest_dict* object, in this case Aphids and Leaf Miners. This is similar to the way a database is set up and the pests class acts as a relational database linking the Plant List and the Pest Dictionary. This makes the new database diagram look like this:




```
class DatabaseTest(unittest.TestCase):
    def setUp(self):
        self.PlantList = PL

    def test_object(self):
        self.assertIsInstance(self.PlantList[0], po.Plant)
```

Now I needed to test if the database worked. In order to do that I added more tests to my testing files, setting up a new test class called DatabaseTest. The separation of DatabaseTest and CalculationsTest meant I could test different aspects of the code without worrying about others. For the database testing I imported the plant list as PL from Database.py, the first thing I did was set up a plant list to play with in the test class and then I set up the *test_object* method. This tested the type of object the data in the list was, this was in case the database entries were pointing to something different that was not the plant class. This passed using the *assertIsInstance* method of unittests. Now I needed to test that the relationships were working for my databases, I did this by creating the *test_pest* method which made sure that the *plant.pestlist* attribute worked and pointed to values returned from *the pest_dict*.

```
Ran 1 test in 0.004s

OK
```

```
def test_pest(self):
    firstpest = self.PlantList[0].pests.list[0]
    self.assertEqual(firstpest, po.pest_dict[0])
```

Test_pest worked in two ways, if it passed it affirmed that the *Plant.pests* attribute contained a Pests class that had a list in it and secondly it confirmed that this was the same value as what should be in the *pest_dict*, meaning that the indexing was correct, since the first item of the pest list of the devil's ivy was the first item in *pest_dict* all the indexes are 0. This test passed meaning that the relationship between the pests and the plants worked as it should and therefore gave me the all clear for reformatting the code in order to compensate for this new plant table.

Refactoring The Code

Firstly, I had to change the calculations code because this was where most of the code pointed to when accessing the plant list, this was easy since all I had to do was import it from Database.py and into the *self.plantlist* attribute in the PlantSort class.

```
from Calculations_Code.Database import PlantList as P


class PlantSort:
    def __init__(self, userinputs):
        self.PlantList = P
```

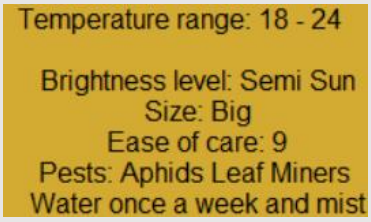
Then came the harder refactoring, the GUI; this was more complex since I had changed the interaction the GUI had with the PlantList in three ways, changing where it was stored, adding more attributes and adding more plants.

Changing where it was sorted was easy since most of the references to the plant list came from the calculations file, the only thing I needed to change was the initail plant list which I just imported as PL and stored in a variable.

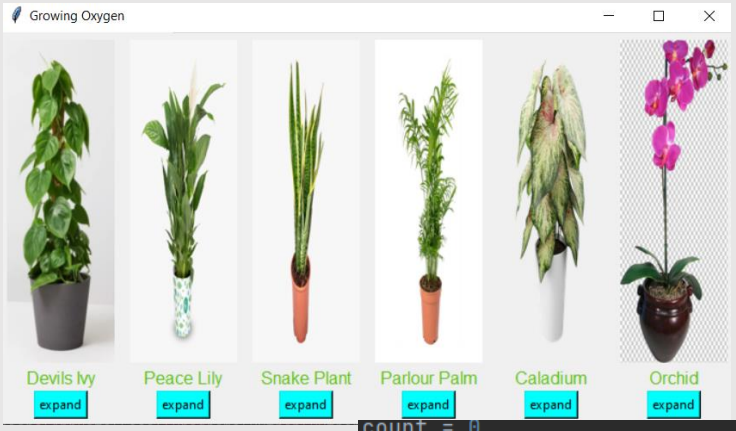
```
Plantlist = PL
```

Now adding more attributes wasn't too difficult, all I had to change was how the Plant Boxes was formatted. I simply added two extra lines, a line of space for the Pest List and one for the care info. The care info was easy since it was just a flat string display. It was the same as displaying the other strings such as the name or temperature but the issue was with pests and how to display it since the pest list could be varying sizes. The solution to this was in the Pests class by creating the __str__ dunder method which was called when the pests object was printed into a string and returns a custom string to print or use. This very simply just returned the Pest list attatched to a string using the .join method for a string and list. The result of all this in the GUI was for the expanded box (or box phase 2). This didn't change much except the bottom lines are the new lines added.

```
def __str__(self):
    return f'{"".join(self.list)}'
```



Refactoring the plant container for a greater number of plants was the hardest part. Previously I had been packing the Plants left as they iterated through the list since this was the easiest way to do it but then I realised that with a larger databse, there would be just a window of increasing width, meaning that I had to go into another column. Running the GUI confirmed my suspicions as I was greeted with this:

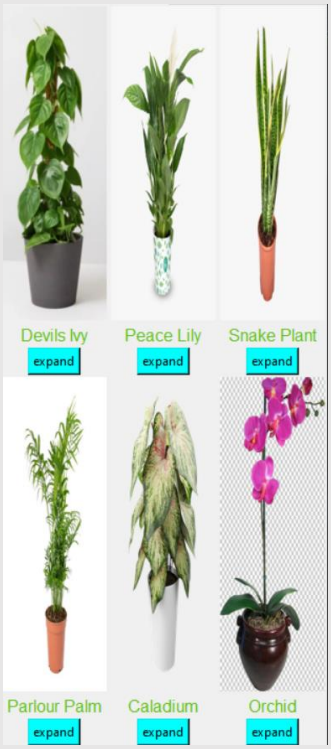


Growing Oxygen, Nic Harrod, Car

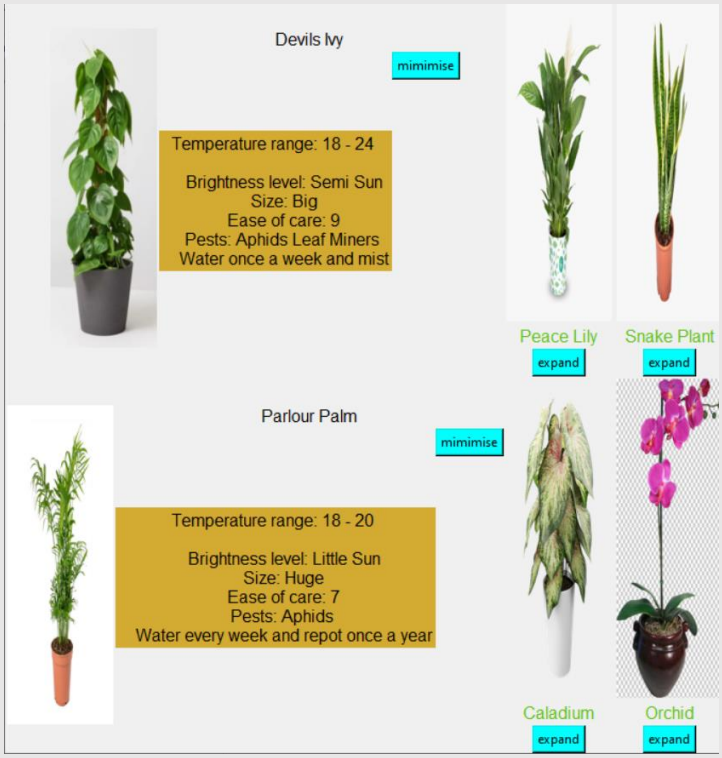
This was far too long and would just get longer as the plants went on, initialy I

```
count = 0
self.frames = [PlantBox(self, plant) for plant in Plist.PlantList]
for pb in self.frames:
    pb.grid(row=count // 3, column=count % 3)
    count += 1
```

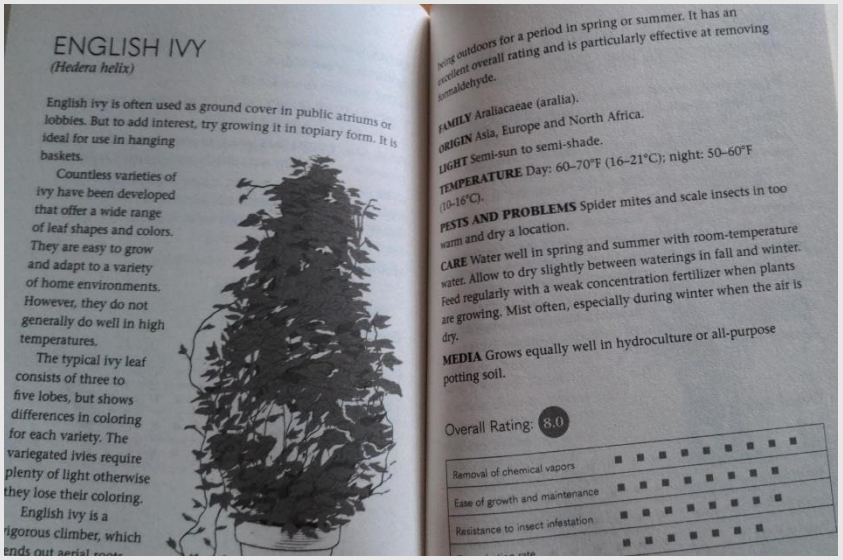
tried to fix this by changing how it would pack depending on it's position in the plant list, but this led to weird formatting and stacking so I decide to change it from a .pack system to a .grid system similar to how the user input menu was formatted. This very simply used a count variable within the for loop to determine the box's postion in terms of column and row, the row was calculated by taking the amount of times the count went into the amount of boxes I wanted per row (rounded down). In this case 3, so if the count was 1, it goes into 3 no times so is on row 0 but if the count is 4 it goes into row 1. For column the postion was decided by how close the count for the plant was to the next lowest multiple of 3, if the count was 1 it was 1 away from 0 so in column 1 but if the count was 4 it was the same since 4 is 1 away from 3. This resulted in a compact stacked Window which was far nicer for goinng into expanded boxes.



This can be shown by the intial plant container on the left and then the expanded boxes if the user wants to compare the attributes of the Parlour Palm and Devil's Ivy on the right. Even though two plants are expanded, the width of the window only increases by one unlike the pack method which would have increased the width by 2.



Now I needed to use the book *'how to grow fresh air'* by B.C. Wolverton in order to appropriately flesh out the databse.Before this I had been creating plant objects that gave a big enough range to show how the code could vary but now I needed to be precise, in doing this I also added more pests to the pest_dict. Now the updated plant box looks very similar to the page to the right in terms of stats for the plant.



ENGLISH IVY
(Hedera helix)

English ivy is often used as ground cover in public atriums or lobbies. But to add interest, try growing it in topiary form. It is ideal for use in hanging baskets.

Countless varieties of ivy have been developed that offer a wide range of leaf shapes and colors. They are easy to grow and adapt to a variety of home environments. However, they do not generally do well in high temperatures.

The typical ivy leaf consists of three to five lobes, but shows differences in coloring for each variety. The variegated ivies require plenty of light otherwise they lose their coloring.

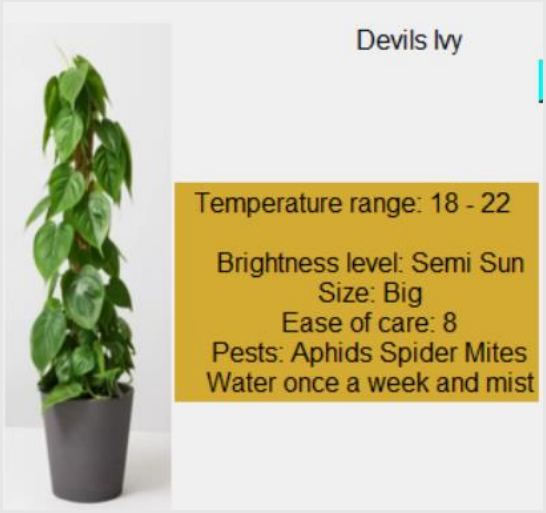
English ivy is a vigorous climber, which

being outdoors for a period in spring or summer. It has an excellent overall rating and is particularly effective at removing formaldehyde.

FAMILY Araliacaeae (aralia).
ORIGIN Asia, Europe and North Africa.
LIGHT Semi-sun to semi-shade.
TEMPERATURE Day: 60–70°F (16–21°C); night: 50–60°F (10-16°C).
PESTS AND PROBLEMS Spider mites and scale insects in too warm and dry a location.
CARE Water well in spring and summer with room-temperature water. Allow to dry slightly between waterings in fall and winter. Feed regularly with a weak concentration fertilizer when plants are growing. Mist often, especially during winter when the air is dry.
MEDIA Grows equally well in hydroculture or all-purpose potting soil.

Overall Rating: 8.0

Removal of chemical vapors
Ease of growth and maintenance
Resistance to insect infestation



I then continued this with all my plants and even added a 'None' option for pests when the book stated that a plant had no pests, in this case it was for the snake plant in which I just passed an empty list into the pests attribute and it returned none in the GUI.

Databases (Continued) Criteria Fulfilment and Review

Below is the table for the criteria that the implementation of new database system completes.

R3	R5
Have a database system which contains multiple tables which the program accesses to display.	Characteristics shown include things like temperature, ease of maintenance, care instructions, size and what the plant looks like.
This is the fundamental of what this part of development is doing so whilst not finished yet, it certainly sets the program up the fulfil it.	By setting up the different fields of the objects as well as inclusions such as the size and brightness dictionaries. The database and GUI are now fully functioning together.

Code Annotation

In order for future mainitnenece of my program to be understandable and easy for others in the future who may work on it, the code has to contain sufficient comments to show what the various algoritihms do. Whilst this document may be a good guide for anyone developing the program, the existence of comments can help anyone understand their way around the code. This section contains all the parts of the code which aren't already shown to be heavily commented such as the *PlantApp* function.

```
def plant_reccomendation(self):
    #setting up all the series identifies of the radio buttons
    var1 = tk.IntVar(self)
    var2 = tk.IntVar(self)
    var3 = tk.IntVar(self)
    # Label for the buttons
    self.templabel = tk.Label(self, text="What is the temperature \n of the location?", bg
    self.templabel.grid(row=2, column=1)
    # buttons for setting temperature
    self.tempbutton1 = tk.Radiobutton(self, text="18-19°C", bg='#826644', value=1,
                                      variable=var1, command=lambda: GOxygen.settemp(18))
    # PLacing the button into the grid
    self.tempbutton1.grid(row=2, column=2, padx=1, pady=3)
```

For each part of the code where there were many parts that did the same thing, for example the buttons and labels in the *UserInputMenu* frame were all very similar so I only commented on one as seen to the left.

Another thing that needed to be explanatory is the default set up for most of the frames, for this I commented through the *Menu* class as seen below, describing what the *self.master, self.pack()* and *self.create widgets()* were.

```
class Menu(tk.Frame):
    def __init__(self, master=None):
        #sets up the master for the frame
        super().__init__(master)
        self.master = master
        #packs the frame
        self.pack()
        #runs the function for packing the widgets into the frame
        self.create_widgets()

    def create_widgets(self):
        #creates a button to show the next frame of the GUI
        self.plant_reccomendation_choice = tk.Button(self, text="plant recommendations",
                                                     command=lambda: self.master.show_frame(
                                                         self.master.frames[UserInputsMenu]))
        #packs the button into a grid
        self.plant_reccomendation_choice.grid(row=2, column=4, padx=10, pady=10)
```

Now the Plant container had to be further annotated in order to explain what the difference between the two branches was as well as the reasoning behind it. This is shown below.

```
class PlantContainer(tk.Frame):

    def __init__(self, master, Plist=None):
        super().__init__(master)
        self.master = master
        #setting up the list of box frames depending on if the new plant list has been passed through
        if not Plist:
            self.frames = [PlantBox(self, plant) for plant in Plantlist]
            #packing the frames
            for pb in self.frames:
                pb.pack(side=tk.LEFT, padx=5, pady=5)
        else:
            count = 0
            self.frames = [PlantBox(self, plant) for plant in Plist.PlantList]
            for pb in self.frames:
                #making the display of boxes neater and not just one line
                pb.grid(row=count // 3, column=count % 3)
                count += 1
```

I then commented through the *PlantBox, BoxPhase1* and *BoxPhase2* classes since they were quite complex. These are seen below.

```
class PlantBox(tk.Frame):
    def __init__(self, master, plant):
        super().__init__(master)
        self.master = master
        #sets up frames of the two phases
        self.frames = {BoxPhase1: BoxPhase1(self,
        #packs first phase
        self.show_frame(self.frames[BoxPhase1])
```

```
class BoxPhase1(tk.Frame):

    def __init__(self, master, plant):
        super().__init__(master)
        self.master = master
        self.name = plant.name
        # setting up the image display
        with Image.open(plant.image) as im:
            # making sure all the images are the same resolution so not to mess with the proportions of the boxes
            im_resized = im.resize((100, 300))
            self.image = ImageTk.PhotoImage(im_resized)
        self.imagelabel = tk.Label(self, image=self.image)
        self.imagelabel.pack()

        self.boxtitle = tk.Label(self, text=self.name, fg='#5BC014', font='bold')
        self.boxtitle.pack()
        self.expandbutton = tk.Button(self, text='expand', bg='cyan', command=lambda: self.expand(plant))
        self.expandbutton.pack()

    def expand(self, pl):
        print(f'expand {self.name}')
        self.master.forget_frames()
        # showing the next phase
        self.master.show_frame(self.master.frames[BoxPhase2])
```

```
class BoxPhase2(tk.Frame):

    def __init__(self, master, plant):
        #setting up the dictionaries for the integer values of attributes to indix in and retireve strings from
        bright_dict = {1: 'No Sun', 2: 'Little Sun', 3: 'Semi Sun', 4: 'Bright Sun', 5: 'Full Sun'}
        size_dict = {0: 'Small', 1: 'Medium', 2: 'Big', 3: 'Huge'}
        super().__init__(master)
        self.p = plant
        self.master = master
        self.name = self.p.name
        #setting up the image
        with Image.open(self.p.image) as im:
            im_resized = im.resize((100, 300))
            self.image = ImageTk.PhotoImage(im_resized)
        self.imagelabel = tk.Label(self, image=self.image)
        self.imagelabel.pack(side=tk.LEFT)
        #setting up the big textbox with a lot of formatting
        self.textbox = tk.Label(self, bg='#D3AA32', text=
        f'''Temperature range: {self.p.mintemp} - {self.p.maxtemp} \n
        Brightness level: {bright_dict[self.p.brightness]}
        Size: {size_dict[self.p.size]}
        Ease of care: {self.p.ease}
        Pests: {self.p.pests}
        {self.p.careinfo}''', font=(44))
        self.textboxtitle = tk.Label(self, text=self.p.name, font='bold')
        self.textboxtitle.pack(side=tk.TOP)
        self.textbox.pack(side=tk.LEFT)
        self.minimisebutton = tk.Button(self, text='minimise', bg='cyan',
                                        command=lambda: self.contract())
        #same as with phase 1, just the other way
        self.minimisebutton.pack(side=tk.TOP)

    def contract(self):
        self.master.forget_frames()
        self.master.show_frame(self.master.frames[BoxPhase1])
```

After annotating the mjaority of the GUI I then went through the plant objects file to ensure it was understandable and so I annotated the plant, pest and user input classes.

```
class Plant():
    def __init__(self, name, maxtemp, mintemp, brightness, size, ease, careinfo, pests, ima

        self.name = name
        self.maxtemp = maxtemp
        self.mintemp = mintemp
        self.brightness = brightness
        self.size = size
        self.ease = ease
        # initialising a passed in array into an object
        self.pests = Pests(self, pests)
        self.careinfo = careinfo
        self.image = image
        # score to rank the plants
        self.desirability_score = 0

    def scorechange(self, val):
        self.desirability_score += val
    # creating a repr so the testing is understandable without having to access attributes
    def __repr__(self):
        return f'Plant : {self.name} Score : {self.desirability_score}'
```

The plant class just needed some polishing around the role of the *__repr__()* function and also why the *self.pests* was initialized like it is.

```python
class Pests:
    def __init__(self, master, pestlist):
        # taking master for the repr
        self.parent = master
        # branching if the plant has no pests
        if pestlist:
            self.list = [pest_dict[i] for i in pestlist]
        else:
            self.list = 'None'
        # returning a list if the class is printed like it is in the GUI
    def __str__(self):
        return f'{"".join(self.list)}'

    def __repr__(self):
        return f'{self.parent.name} pests : {"".join(self.list)}'


# dictionary for the pests class to look in
pest_dict = {0: 'Aphids ', 1: 'Leaf Miners ', 2: 'GreenFlies ', 3: 'Spide
```

The pests class just had to show why there was a diffeerence between the *__repr__()* and *__str__()* and also explain the branching which essentially justified the whole existence of the class.

```python
class UserInputs():
    # class to act as a communicator for the GUI and algorithm
    def __init__(self, temperature, brightness, size, ease):
        self.temperature = temperature
        self.ease = ease
        self.size = size
        self.brightness = brightness
    # function that sets the temperature according to a passed value
    def settemp(self, val):
        self.temperature = val
        # testing that it works
        print('set', val)
        print(self.temperature)
```

I then commented through the *PlantSort* class and explained some of the ideas behind the calculations as seen below.

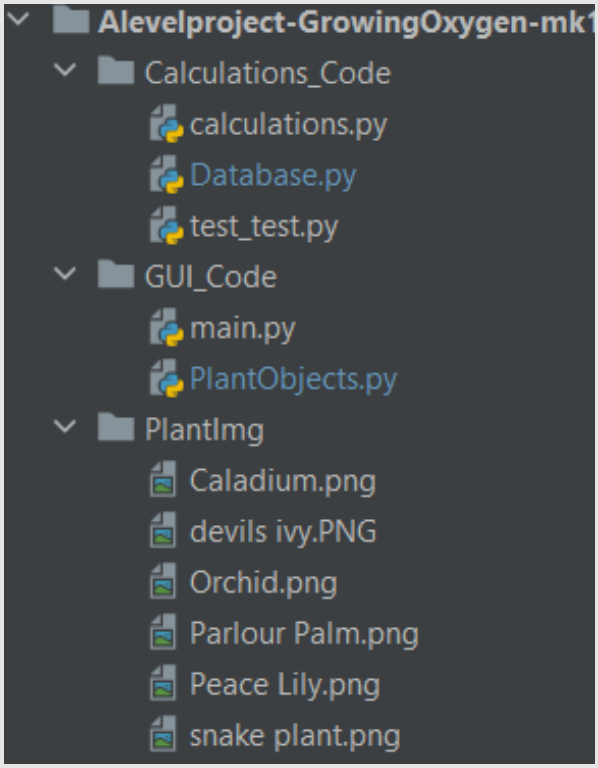
```python
class PlantSort:
    # taking user inputs as a parameter
    def __init__(self, userinputs):
        # creating an internal plant list
        self.PlantList = P
        self.UI = userinputs

    def calculate(self):
        # running all the scoring procedures
        self.score_bright()
        self.score_temp()
        self.score_size()
        self.score_ease()
        # sorting according to the score
        self.PlantList.sort(key=lambda plant: plant.desirability_s

    def best(self):
        # returns the best plant for testing
        self.calculate()
        return self.PlantList[0]
```

```python
    def score_bright(self):
        # checking if the user inputted a brightness value
        if self.UI.brightness:
            # looping through the plant list
            for i in self.PlantList:
                # if bang on the value gets 5
                if self.UI.brightness == i.brightness:
                    i.scorechange(5)
                # otherwise gets less
                elif self.UI.brightness == i.brightness + 1 or sel
                    i.scorechange(2)
                elif self.UI.brightness == i.brightness + 2 or sel
                    i.scorechange(1)
```


er 12610

Tidying The File

Now since the code was finished, I wanted to restructure the file in order to make it clear where different parts of the code were, I did this by breaking it down into 3 sections, Calculations Code, GUI Code and Plant Images, this was so all the code knew where the other parts were and to any onlooker it was clear what was happening. What the file structuring looked like before can be seen on the left and what it looks like now can be seen on the right.



After this I had to refactor the code slightly in order for all the modules I get from other parts of the code to work so they can recognize each other. Using the abbreviation of the file directory they were stored in allowed me to access different modules across folders as seen below in main.py with code from the calculation section.

```
from Calculations_Code.calculations import PlantSort as PS
from Calculations_Code.Database import PlantList as PL
```

Then the refactoring for the plant images was more complex since it didn't seem to recognize just doing '*PlantImg.[PlantName].png*' so I had to copy the full directory into the image calling, another advantage of the database system because I only had to do this once instead of for each instance of the list.

```
r'C:\Users\nicpa\PycharmProjects\Alevelproject-GrowingOxygen-mk1\PlantImg\devils ivy.PNG'),
```

I then renamed some of the objects I was using in order to appropriately match the code, previously I had used the 'David' object as my main plant app instance but then I renamed it to GOxygen in order to reflect the actual contents of the code.

```
david = UI(None, None, None, None)

print(david.temperature)
app = PlantApp(Plantlist[1], True, david)
app.mainloop()
```

```
GOxygen = UI(None, None, None, None)

print(GOxygen.temperature)
app = PlantApp(Plantlist[1], True, GOxygen)
app.mainloop()
```

I then also went through the GUI of the code and removed all the *self.screenwidth* attributes because it turned out they didn't do anything. Below is the before and after of this for the *UserInputsMenu* class.

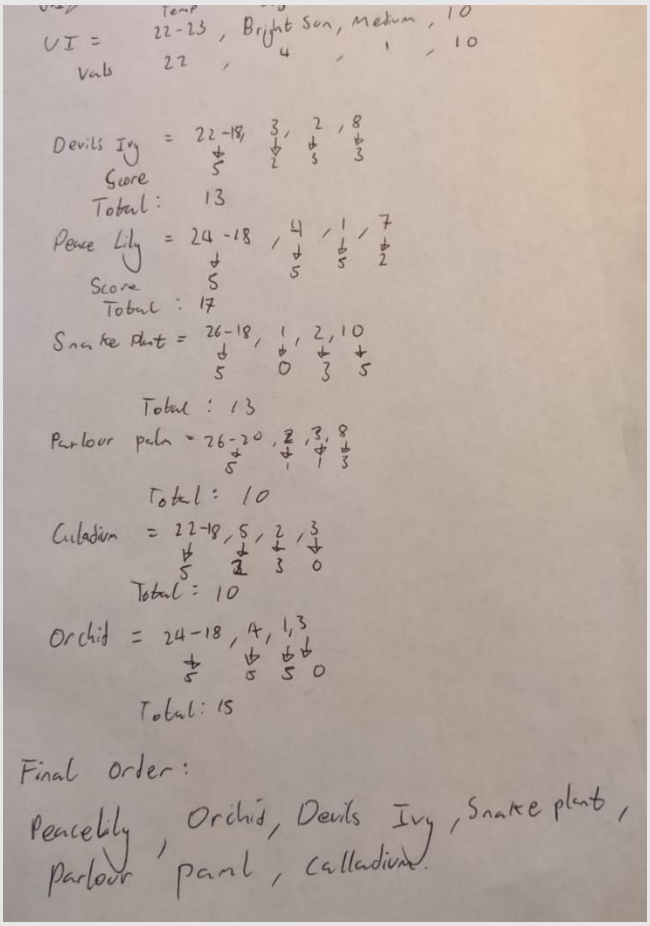
```
class UserInputsMenu(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.screenwidth = self.master.winfo_screenwidth()
        self.screenheight = self.master.winfo_screenheight()
        #packing the frame
        self.pack()
        #packing all the widgets
        self.plant_reccomendation()
        self.master.config(bg='green')
```

```
class UserInputsMenu(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        #packing the frame
        self.pack()
        #packing all the widgets
        self.plant_reccomendation()
        self.master.config(bg='green')
```

Evaluation

Final Run Through and Post-Development Test

To ensure that the code works and showcase what it looks like for a normal run through, I would need to run it on a computer that's not my own and do it according to the post-development test described before to simulate a user actually inputting values. For this I need to initially calculate what the values would be from this. The test asks for the expanding of the first and third plants so those are the two I need to calculate fully. Below are my calculations for the result from the test, each plant is compared to the user inputs and then the final list is produced at the end.



Each desirabily score is calculated and then compared at the end, so in this case the first and third plants will be the peace lily and the devils ivy. This means that the expanded box for a peace lily should look like this :

Temperature Range: 18-24
Brightness Level: Bright Sun
Size: Medium
Ease of Care: 7
Pests: Mealybugs Leaf Miners Spider Mites
Water when dry and clean leaves

Here is the video for the final run-through:

Post-development test

As seen all the visuals were correct and worked as expected, which is good. Now I needed to check the variables after the calculations to ensure the scores were correct. Below are the expected and calculated scores for devils Ivy and Peace Lily. Alongside it are the variables able to be seen in the python console. The values for score match up meaning the program worked.

Plant Name	Expected Score	Calculated Score
Peace Lily	17	17
Devils Ivy	13	13

Future Development

Self-Response

The first stage of analyzing how the project went is by understanding where I felt I could have improved. First and foremost, the project as it is currently a prototype, it's not expected to fulfil everything the

design and stakeholders entailed. Things that I would do in the future would include accommodating further for Erin's design since that is more aesthetically pleasing and marketable.

Another thing to do in the future is to implement a SQL database fully, I felt that whilst developing the program I had run out of time in doing the GUI and so didn't want to push myself for time when making this write-up. Because of that I ended up using a similar data structure as I did for my temporary plant objects. A SQL database is ultimately far more convenient than what I was using, although it takes some time understanding and setting up, once it's done I would be able to easily retrieve and update objects. This means that the Plant class I had used before would be obsolete since each record in the Plant table would have been able to be treated and accessed like an object anyway, even with updating them. This is because the plant object isn't particularly complex other than with the desirability score and pest adding. The score can simply be updated on the table and the pests would just work via a link table.

The final thing I would add in the future would be the plant calendar concept. This was always an unlikely thing to happen, but it was an interesting concept since it brings in the idea of updating a database as the program runs. It would work by having the user log in to the main menu and either view their saved plants on a calendar or add to their saved plants from the plant recommendations.

Stakeholder Response

I then interviewed my stakeholders for what they thought about the program. I was able to get an interview from Erin and I was then able to get an interview of what Matthew thought. I think these interviews went well because they showcased what the differences of the stakeholders is.

Other Response

I also gave the program to some teachers and others around me. They had quite similar responses to it as the stakeholders did.

- I first asked my classmate Ted to work through it and he said that he enjoyed the program but had an issue with the fact the images were all different aspect ratios and had backgrounds.
- I then asked a family friend Aaron to run through the code and his issue was that the user wasn't given enough information about what was going on, meaning that in the future I need to stive to make the code more accessible. Especially with the care slider in the inputs since most people didn't think that was explanatory enough.
- After I gave it to another one of my classmates, Vincent. He is a computer science student so understood quite a lot of it. He had an issue with the plant boxes not being ordered and didn't like how stretched the images were.
- Finally, I gave it to one of my computer science teachers, she said that the images were a bit too stretched and it would have also been useful to have a back button, so the user doesn't have to run the code again if they want to change anything. Another issue she had was that there wasn't a higher or lower temperature so users with cold or hot rooms wouldn't be able to do it.

From these I am able to conclude that in the future I would add a back button which can just reset the user inputs and scores and load the user input menu again. I am also seemingly not very explanatory in the program so I would add numbers for the plant boxes to display what order they go in. I would also describe my ease slider better because a lot of users seem to take 10 as being hard and 1 as being easy when it is in fact the opposite. Finally, I would flesh out the database far more and add many more

values; instead of the same six being displayed each time it would be the six most relevant from a databank of 20.

Success Criteria Fulfilment

Now I need to fulfil and mark what of my previously stated success criteria I have filled, I will use the key of green for success, red for failure and amber for somewhat fulfilled. There will also be justification for why the requirement is the colour it is.

R1	R2	R3	R4	R5
Create a program that recommends plants based on attributes of the user's room.	Have a user interface which takes inputs from the user and then does the calculations for the recommendation.	Have a database system which contains multiple tables which the program accesses to display.	Display the characteristics of the recommended plant and what they look like.	Characteristics shown include things like temperature, ease of maintenance, care instructions, size and what the plant looks like.
This is the basis of the program so if this wasn't achieved then the program wouldn't be finished.	Although not as explanatory as it could be, I would say that I fulfilled this criterion to the standard set out.	Whilst I have a database system, it isn't particularly advanced due it just being dictionaries and arrays.	I think this went really well with he expanded plant boxes especially since it means that the user can choose want plant they learn about.	I feel like I kept the information simple but still provided enough to not make the recommendation completely useless.

D1	D2	D3	D4	D5
Add a plant saving function unique to the user potentially based on a log in.	The database is made using SQL.	Use an image storing system using BLOBs (Binary Large Objects).	Have a calculated ranking that is based off the user inputs.	Whilst needing to look good the GUI also must be simple enough to understand in order to be appropriately efficient.
This is something that I would have liked to do but ran out of time for it.	I also ran out of time to revisit SQL for my database and the code was already working well enough with the temporary database.	BLOBs only work with SQL database so there was no feasible way for me to implement them.	While I would have marked this criterion as green because the plants were ranked according to the user inputs, Vincent raised the point that there needs to be a numerical display of the ranking so that's an issue for the future.	The GUI has remained simple whilst still able to communicate the data required and presented in it quite well.

Usability Review

In order to review usability, I have to first do what I set out in my design section. I will then mark each of accessibility, clarity, engagement and error tolerance out of 10.

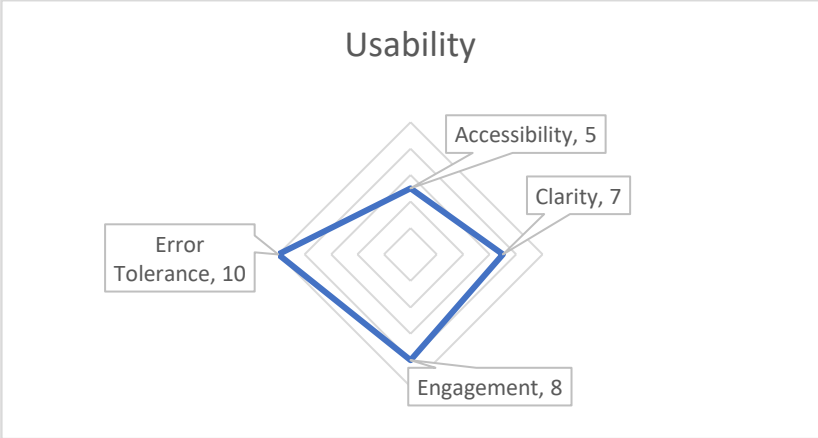
The first of these was usability so asking different individuals about the program and what they thought of it, this has been exemplified above in the other responses section. In conclusion the accessibility of the program is quite high but there is an issue with the explanation of what the program does. This is because many didn't seem to understand the direction the slider for ease went, this means that in terms of accessibility and understanding I am going to mark myself a 5.

In terms of clarity, this is for ensuring that the user knows what's going on at any given moment. I feel like the fact I have marked the User Inputs with radio buttons boosts my score on this, but another issue is something that Vincent pointed out, the user is unsure which direction the rankings go for the Plant Boxes. Because of this I am going to mark my clarity score as a 7.

For engagement, I was able to measure this by seeing how Erin reacted to the program while running it. This is shown by this video of her using the program. As seen from the video she seems quite engaged in the program and her feedback also mentions this fact where she says that she enjoyed the aesthetic and different colour contrasts of the program. This means that the engagement of the program is sufficient and in my own view is quite good, an issue raised though is the amount the images are stretched. This is a problem because it breaks the focus of the user by distracting them from the actual content of the program. A remedy for this would be to ensure that all the images are the same resolution at the start of the program. Because of this I am going to score the engagement of the program as an 8.

Finally, is for error tolerance. The user inputs all take discrete values and as it stands currently, the program can't error out from anything the user does. If they forget to enter in an input or don't know it, the calculations can accommodate for that by branching when None for a user input is passed in. The Plant Boxes are also all able to be expanded at the same time so the user can't overload the program with two expansions at once. Because of this I am going to score Error Tolerance as a 10.

Below is the radar graph for my program in terms of the usability scores.



Limitations

Future Limitations

In the future for when development continues, the projection of what to do has to be clear. This is because there are limits to how far the solution can go, there are feasible developments of the solution such as the plant calendar and database structuring but then there are things the solution can't solve.

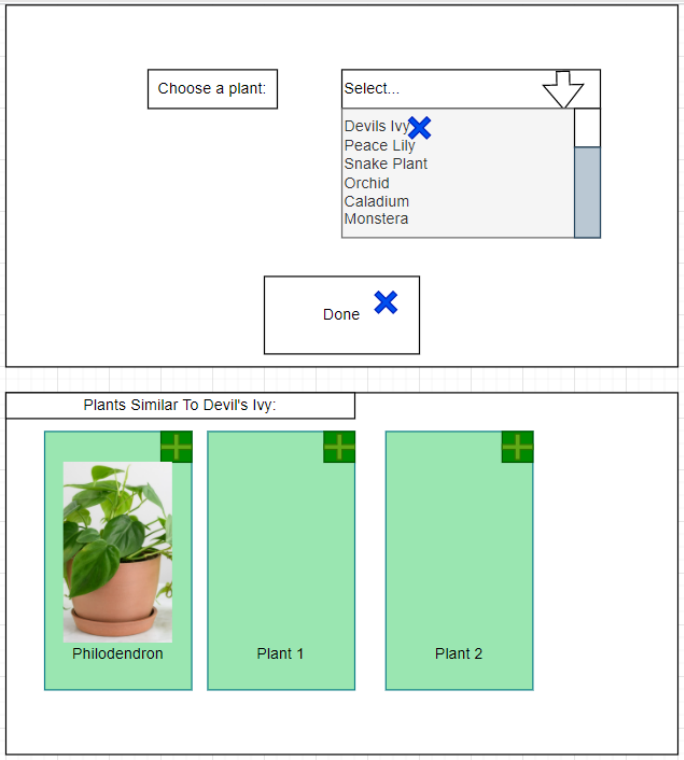
One of these things is the fact that the program will never be able to buy plants for the user, whilst it is possible for it to give links to the user to click and go to a website and buy the plant. The program will never process payment itself because that is straying too far from the core of the program and also adds some unnecessary payment security which would take too long.

In the same sense another limitation is networking of the program, when the plant calendar and user login is added. The issue with this is that there would have to be a server hosted databank of logins which would be too complex so it'd be better to just have users saved locally instead of spending time creating a login system and like before, would need to have heightened security of the program using things like hashing.

Removeable Limitations

There are also some restrictions and limitations on the current program that can be lifted with future development.

One of these liftable limitations is the lack of a database in the system. In the future when a database is added, the GUI would be able to send queries to retrieve plants that are far more advanced than just the most recommended plant. An example of this is searching by plant, the user selects a plant and then the system retrieves plants similar to it, an example of this is below.



The user can select a plant from a dropdown of a large series of plants and then a query will retrieve a series of plants similar to it and display them in the GUI just like it did for the original code. This would also mean a more fleshed out main menu as the options for what the user can do with the plants increases, the system becomes less a plant recommendation app and more a plant maintenance and help app.

Another changeable limitation would be weighting the different values of room attributes more, for some plants they can survive in conditions that aren't their ideal better than others and for some they have attributes that aren't especially relevant to them surviving. For example, the ease of care should be weighted more so that the user gets what they want. An orchid is a very difficult plant to care for but if the rest of the user's room aligns with its attributes, then it can still get recommended. This seems counter intuitive for my program to be catering to those with different needs so the calculation should prioritize matching the ease of care of a plant more than anything else.

Future Maintenance

In the future for when development continues, the projection of what to do has to be clear. This is because there are limits to how far the solution can go, there are feasible developments of the solution such as the plant calendar and database structuring but then there are things the solution can't solve.

The largest issue the program currently faces is the lack of depth that exists within many of its components. The GUI is quite simple and bland, I think with future maintenance of the current solution that could be fixed to result in something like what I produced in my design section. Another example of the lack of depth is the database and fact that there aren't many plants in it. An advantage of using the database is that the fields can easily be filled out from an Excel spreadsheet, this means that the maintaining and improving the data in the system is far easier, the GUI just has to be slightly tweaked to accommodate for this, such as changing the image sizes and the dimensions of the grid of the boxes in the plant container.

The annotations of the code are quite in depth, this means that anyone picking up the code will be able to understand it and change the code. This will probably be mostly used in the calculation part of my code because that can be changed far more in depth with weighted scoring.

Final Code

The link below is the code to my GitHub repository which has the files containing all my code.

https://github.com/nichar07/Alevelproject-GrowingOxygen