Year 13 Programming Assessment

Plan and write code for a complex computer program for the following task. The program should use objects to store appropriate information, and a GUI allowing access to the programs functions.

You must spend **10 timetabled classes and four hours of homework for planning, write code, testing and documenting**. You may choose one of two briefs given below – ICCT20 World Cup Ticketing system or Science Quiz

## Brief 1 – Ticketing app

You will be designing and implementing a software for ICC T20 World Cup ticketing system. The program keeps tracks of available tickets, plus show the total value of tickets sold.

## Specification

The ICC T20 World Cup Committee releases limited amount of tickets from time to purchase for the games

|  |  |  |
| --- | --- | --- |
| **Event** | **Tickets available** | **Ticket Cost** |
| Opening Ceremony | 100 | 100 |
| Table A and B matches | 100 | 50 |
| Semi Final 1 | 100 | 150 |
| Semi Final 2 | 150 | $150 |
| Finals | 150 | $250 |
| Closing Ceremony | 200 | $200 |

* The interface should display how many tickets are available for each event, out of how many total can be sold on a given event (*Example: Semi-Finals - 54 available/150 released for sale*)
* The interface should allow the users (user input) for a selected event.
* If any details change at any point, the interface should update to reflect these changes.
* The interface should also display:
  + The total amount of tickets sold (*Example: 335 tickets sold today*)
  + Total value of tickets sold (*Example: $2987 earned today*)
* Error checking and prevention should be implemented as much as possible.

You will need to maintain a **daily/testing log** that documents the development of your programme.

## Brief 2 – Science Quiz app

You will be designing and implementing a Science Quiz app for primary school students aged 9-13 years. The app keeps track of details of multiple use.

## Specification

* Greet user and give brief description of quiz
* Give instructions to enter name, age and difficulty level
* Display series of questions and prompt user to answer questions
* Give feedback and award points for correct answer
* At the end of quiz, give them a report with username, age, difficulty level and points
* There is an option for users to go back and complete the quiz several times.

You will need to maintain a **daily/testing log** that documents the development of your programme.

## Appendix 1: Planning Guide

### Task 1: Identify user inputs

*What program methods can the user trigger through the interface?*

|  |  |
| --- | --- |
| Login username | The user can input their username to login to the program |
| Login password | The user can input their password to login to the program |
| Login button | The user can press the button to proceed to the next screen if login is successful, otherwise an error message will pop up |
| Login error | A message box telling the user they have an incorrect username or password or that they left a field empty. User has to press ‘ok’ to continue |
| Register button | Will take the user to the register screen from the login screen |
| Register username | The user can input their desired username for the program |
| Register password | The user can input their desired password for the program |
| Register confirm button | The user can press the button to proceed with the registration. If successful, the user will be taken back to the login screen, otherwise a error message will pop up |
| Register error | A message box telling the user they left a field empty or their username has already been taken. User has to press ‘ok’ to continue |
| Logout button | Button will ‘log’ the user out where it returns the user to the login screen and wipes current user data to free memory |
| Intro screen proceed button | Takes the user to the next screen, the select screen |
| Select back button | Takes the user from the select screen back to the intro screen |
| Select radio buttons | Four radio buttons with the different subjects the user can take the quiz on |
| Select proceed button | Takes the user from the select screen to the quiz screen while passing on the selected radio button data to the quiz screen class |
| Quiz entry | The user can input their answer to the question on screen |
| Quiz submit button | The user can submit their answer to go to next question or results screen, but an error will be called if they leave the field empty for quiz entry |
| Quiz error | A message box telling the user the left a field empty. The user can press ‘ok’ to proceed |
| Results proceed button | Takes user back to the intro screen from the results page |

### Task 2: Identify information to be displayed

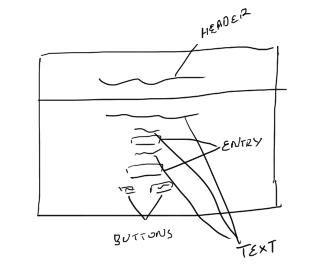
*What information will the interface need to display to the user?*

|  |  |
| --- | --- |
| Program instructions | Displayed on the main menu, it will tell the user how the program reads input as typing a mathematical answer through only keyboard input is quite hard due to a lack of symbols |
| User history | Also displayed on the main menu, it shows the user’s past scores. This is displayed in the form of a treeview. |
| Quiz question | Displayed on the quiz screen, it shows the question the user needs to answer |
| Results | Shows the user their results out of 5. |

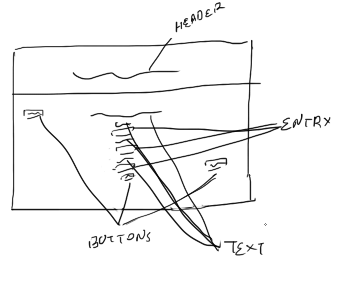
### Task 3: Sketch interface design

*Draft a rough design for the interface that allows the user to trigger functionality in task 1, while also annotating where the information in task 2 will be displayed. UML class diagrams and flowcharts*

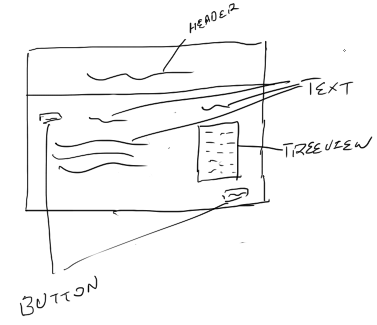
Login screen



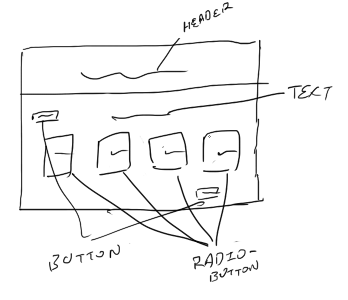
Register



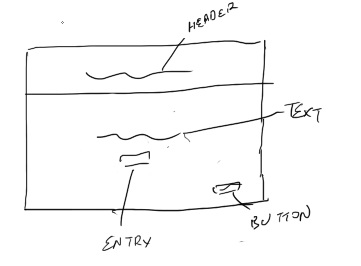
Main menu



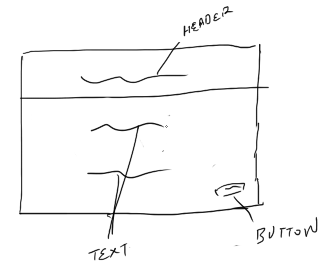
Select menu



Question



Results



### Task 4: Identify any classes required

*Explain what the class will represent, plus listing what information will be stored in the class and any methods the class will have.*

|  |  |
| --- | --- |
| App | Inherits from tk.Tk, becomes the root class of the GUI. All other frames will be gridded onto this class as it is the main window itself. The class will contain data for the current user and current screen. As this class is the root class of the GUI, it will also set the title to ‘Level 3 Calculus Revision’ and geometry to 540 by 960 (1/4 of a 1080p 16:9 screen lengthwise). It binds the enter/return key accordingly to the current screen to a button that will allow the user to proceed through a series of conditionals. This binding is the only method present in this class. |
| Header | The header is a class instance where it will stick on to the top row of the root window for the whole duration of the program. The header will just have a title ‘Level 3 Calculus Revision’. This class does not contain any methods and instead just has the ttk style options for the label. |
| IntroScreen | The intro screen is the class for the login frame. This frame will just have two entry  boxes, one for username and one for password entries. The frame will have two labels describing which entry box is for which, as well as a button to register an account and one to sign in. The frame will contain the function to check the user’s credentials to see whether it matches with the database records. Like the other classes that inherit from tk.Frame, it will contain a method to grid its own children widgets. |
| Signup | The class for the register frame. This frame has three entry boxes, one for the name of the user, one for their desired username and the last one for their desired password. There are also three buttons that the user can interact with. The back button will take the user back to the login screen which the user may wish to use if they mistakenly pressed the register button or already had an account to use. The show password button will reveal the text entered into the password entry box for if the user wants to double check their password. Lastly the register button is to submit the registration and will take the user back to the login screen once registration is successful. This class will also inherit from tk.Frame and hence contain methods to grid its own children widgets, as well as the registration function to enter the new user into the database. |
| MainMenu | The main menu is the class for the main menu screen. This class contains a text that guides the user on how to use the program, as well as a treeview that shows the user their previous scores. Above the instruction text, there will be a simple welcome for the user. For interactable widgets, the user is able to scroll through the treeview if they have many scores submitted into the database. There are two buttons that the user can also interact with. The logout button will clear the system of current user data and take the user back to the intro screen where they can log back in if they wish to do so. This is particularly useful for if the user is logged into the wrong account or wishes to switch accounts. The proceed button however, can take the user to the select menu where they will be able to select the standard they wish to revise. This class also inherits from tk.Frame and will contain the method to sort the treeview, the static logout method, as well as gridding its own children widgets. |
| SelectMenu | The select menu is the class for the selection menu that the user can interact with to choose the standard they wish to revise. For interactable objects, there will be radio buttons for the selection processes, as well as a back and proceed button. For the four radio buttons, there will be one for each standard available (from the list of surds, complex umbers, polynomials or differentiation), and each of the radio button will span the whole row on the frame. There will be no indicator for the radio button to make the above work. At default the first radio button will be selected to prevent empty selections. The back button will take the user back to the main menu in case they want to reread the instructions or have a look at the treeview again to motivate themselves, or logout to revise under a different account. The proceed button will then take the user to the question frame where they will be able to do the actual revision. This class also inherits from tk.Frame and will contain the method to pass selection data to the question frame, as well as gridding its own children widgets. |
| Questions | The question class is for the questions that will appear. This class will work as a container class for the questionobj class which will contain the actual question that the user must answer. In the question class, there will be a widget for the questionobj class where widgets from that class will be used in this main container class. There will also be a submit answer class for the user to submit their answer and move onto the next question or results screen. This class also inherits from tk.Frame and will only contain two methods, one of which is again, to grid its own children widgets. The other one however, will check and make sure the submit answer button will bring the next question or take the user to the results screen |
| QuestionObj | The questionobj class is for the questions that will appear in the question container class. This class will initialise the question by requesting a randomly generated question from the class of the chosen standard. It will have the question text stored as a label widget, with the text taken string method of the class of the chosen standard. There will also be an entry box for the user to type their answer in. The class will inherit from tk.Frame. There will only be one method in this class which is to grid the two children widgets that it will contain. |
| ResultsScreen | The resultsscreen is the class for the results screen that the user will be taken to after completing the revision questions on their chosen standard. The front end will simply be two label widgets, with the first one stating the second one is the results, and the second being the user’s score out of 5. There will also be a return button which is the only interactable object in this frame. This button will take the user back to the main menu screen where they can see their score has been appended to the treeview of their previous scores stored in the database. This class will also inherit from tk.Frame and will contain the obvious method to grid its own frames, as well as the method to calculate the results that the user has achieved, appending it to the database afterwards. |
| User | The user class is a support class for easy parsing of user data from the database. It will contain the user id, name, username and password from the user. This class does not inherit from anything and has no methods of its own. |
| Scores | The scores class, similar to the user class is for easy parsing of the users scores from the database. It will contain the user id, date, subject and score that the user has achieved. This class will also not inherit from anything and will not have any method of its own. |
| Surds | The surds class is for the generation of the surds question for the questionobj class. It will contain three methods. The question\_gen method will generate random numbers for the surds in the range of 1 to 50 inclusive and check to see if these numbers will reduce to the same surd, rerunning the process if it does not. After a viable set of numbers has been generated, it will return the question in a readable format as a string method of the class, meaning when the questionobj class calls for the string method of this class, it will return the readable format. The string method (\_\_str\_\_) has been used over the object representation method (\_\_repr\_\_) as this class will be stored in a list to be referred to later for its last method. The last method of this class is the get answer method which will calculate the answer and return it in the same format that the user is expected to type it in. |
| ComplexNumbers | The complexnumbers class is for the generation of the complex numbers questions for the questionobj class. It will also contain three methods. The question\_gen method will generate random numbers based on the form that is chosen based on another random selection of the two forms available (polar and rectangular numbers). It will then return the generated question in a readable format as the string method of the class. This is for the same reasons as stated in the surds class. Lastly the get answer method will calculate the answer using methods from a core module (a module from the libraries python comes with) called ‘cmath’, which is similar to the math library but deals with complex numbers. The answer is then returned in a similar format to how the user is expected to type the answer in. |
| Polynomials | The polynomials class is for the generation of the polynomials questions for the questionobj class. It will also contain only three methods. The question\_gen method will generate random numbers based on the degree of the polynomial. It will then return these numbers in a readable format as a question for the user under the string method. Again, a string method is used for the same reasons as the surds and complex numbers class. The get\_answer method will index through the randomly generated numbers and calculate the answer, then returning it in a format that the user is expected to return the answer in. |
| Differentiation | The differentiation class is for the generation of the differentiation questions for the questionobj class. It will also only contain three methods. The question\_gen method will generate random numbers and assign these numbers a variable to differentiate with. This is passed on to turn into a readable format as the string method for the class for the same reasons as for the surds, complex numbers and polynomials classes. Lastly the get\_answer method will calculate the answer and return it based on the variable that was randomly selected, as well as parsing the information into a format that the user is expected to return the answer in. |
| DB | The DB class deals with the database connections from the main app class. It will contain four methods, all of which are for the main class to use. The first method is the initialise method, which is carried out if the program is run for the first time. This will insert the necessary tables for the main app - users and scores - with the necessary columns for each. The next method is called ‘fetch’ which will return the requested data for the main class. The add\_user method is for inserting new users from the registration process into the database. Lastly, the add\_score method is for inserting the new scores from the revision to the database to be used for the treeview in the main menu screen. |

### Task 5: Identify any constants or existing data if required

Due to the nature of the program, there is little need for hardcoded data, however, this means that some of the data can be classified as constants of existing data.

* The users that have been registered into the system
* The scores that each user has attained
* The number of variables used in each standard’s question generation
* The information to setup the database in the case of running the program for the first time or on a new system
* The different standards that the user can choose from
* The variables in the generation of the questions of the selected standard where necessary.

### Task 6: Identify indexed data structures

There are quite a few indexed data structures that are involved.

* List for the radio buttons used in the selection menu
* List to contain the class of each questionobj that has been called
* Another list to contain the user’s answers
* List to contain the current user’s details
* List to contain the current screen
* List of the users entered into the database
* List of the scores entered into the database for the existing user
* List for the surds generated
* List for the simplified forms of the surds generated
* List for the forms of complex numbers
* List for the divisors of the polynomials
* List for the dividends of the polynomials
* List for the quotients of the polynomials
* List for the different variables for differentiation

### Task 7: Determine what calculations are necessary

*Write out the calculations the program will have to compute.*

As the nature of the program requires, there will be on calculation necessary in the main routine, while the rest are in the question generation classes.

* The user’s results used in the main routine. This will grab the lists from the question class and the results class, the former being the class instances for each generated question, with the latter being the user’s answers. The lists will be run in parallel, which precautions has been put in to make sure there is no mismatching of the indexing throughout the two lists. The string answer calculated from the generated question’s class will be compared to the user’s answers using string relativity done through the integrated sequence matcher class (a core module that comes with python). If the relativity is higher than 75%, the answer will be accepted as correct and the user’s score will be incremented by one.
* In the surds class, the generation of the question will be done through the random module integrated as a core component of python. There will be a random generation of 2 numbers, both with the range of 1 to 50 inclusive.

A check will be carried out for if the surd number (one of the 2 numbers initially generated at the start of the method) can be simplified, rerunning the function if it cannot. This check is done by checking if the square of every number - from the number’s square root, to 1 – divides perfectly into the surd number.

The answer generation will only have to calculate the sum of coefficients. The simplification will be done in the question\_gen method during the check. The results from this are appended to a list making it not necessary to do any further calculations in the answer generation module.

* In the complex number’s class, there will be a random selection between two forms – polar and rectangular. If polar form is selected, two random numbers will be generated – one for the absolute value/modulus, and one for the angle in range from negative pi to positive pi. Uniform selection is used for this to prevent decimals.

If rectangular form is selected, two random numbers will be generated, one for the real part, and one for the complex part of the numbers in the range of -5 to 5. Uniform selection is used to prevent decimals.

### Task 8: Develop a modular structure for your program

*Describe any methods that the computer program will have, identifying any sub-methods where required.*

There are many methods listed in each class, with only a few in the main routine. Starting with the main routine:

* Update scores where the user list is cleared and a new set of data retrieved form the database is appended to the list if the database does not return an empty list.
* Screen change where the current screen is forgotten and a new screen is gridded, also calling its grid frames method to grid the children widgets of the new screen’s frame.
* Proceed function for the app class where the screen data is taken and sorted through for when the user presses the enter key.
* The various grid frames methods for every class that inherits from tk.Frame. This method is used in the screen change method in the main routine to grid the children widgets of the class. This will not be mentioned in any of the classes that does contain it to avoid repetition.
* Login method for the intro screen class. This method will check for if the username matches with the systems, moving onto the password is it does. It will then hash the password entered and try to match it with the one stored in the database, clearing the inputs if the hashed passwords do match. The user data is then appended to the user list in the main routine, the treeview configuration function is called and screen changes to the main menu.
* Back method for the registration class. This just changes the screen from the registration screen to the intro screen and erases all user inputs in all the input boxes available (name, username and password entry boxes).
* Register method for the registration class. This will check if any of the three input boxes are empty when the user presses the register button and returns an error if any of them are empty. It will also display an error for if the username has already been taken. Lastly if all the above conditions are met, the user is added to the system and the back method is called.
* Treeview configuration method for the main menu class. Configures the newly created treeview with the correct options to load on the data correctly.
* Treeview insert method for inserting the scores into the treeview widget created in the main menu class.
* Logout method for the main menu class. Will clear current user data from the temporary memory and change screen to intro screen.
* Proceed method for the select menu class. Will change the subject variable on the question class and call screen change to the question frame.
* Frame check in the question class. To check if the user input from the questionobj class is empty, returning an error if it is, and the user has pressed the submit answer button. If this check is passed, will check if the amount of questions is at the cap (pre-set to 5), passing the data onto the results class and changing the screen to the results class, else calls its own grid frames method.
* Get results method in the results class. Will compare the calculated answer with the user answer and add to the score counter if 75% of it matches. Appends the score to the database afterwards.

For the proceeding methods, they all serve the same purpose for the question generation classes.

* Question gen method will generate the numbers necessary for the questions of each standard. This may also select from a pre-set list where necessary as part of the question generation.
* Get answer method will calculate or get the answer to the question, returning it in a form that is similar to the way the user is guided to write it in.
* String method to parse the question data into a readable format for the user to be able to interpret it.

Lastly the methods used in the DB class for the database queries.

* Initialise method to setup the tables for if the database is empty
* Fetch method to get the data requested from the database and return it in the 2d list format.
* Add user method to insert a newly registered user into the database
* Add score method to insert a newly attained score into the database.

### Task 9: Define the methods identified

*Describe the methods for both the main program and any classes in terms of input and/or output where required. You may choose to do this with flow charts or pseudo-code (not Python code!). Add in additional steps or explanations using sequential, conditional, iterative statements where required. Identify global and/or local variables.*

The following methods were used in the main screen:

* Update users method

clear users list

fetch login credentials from database

for user in login credentials

append to users list with support class

* Update scores method (very similar to update users)

clear scores list

fetch scores from database

for each score

append to scores list with support class

* Hash passwords method

return sha256 encoded version of password

* Screen change method

forget current screen

set current screen to new screen

grid new screen

run new screen’s grid frames method

The following variables are all global as they are declared in the main routine

* db = DB class
* app = App class
* header = Header class
* intro screen = Introscreen class
* register screen = Signup class
* main menu = Mainmenu class
* select menu = Selectmenu class
* question = Question class
* results = Results class
* current user = empty list ()
* current screen = list with intro screen
* users = empty list ()
* scores = empty list()

For the main methods in each of the main classes that deal with the front end:

* Init method in the app class

Set title to ‘Level 3 Calculus Revision’

Set geometry to 960 by 540 at top left of screen

Bind enter/return key to proceed method in the app class

Set column weight

Set row weight

* Proceed method in the app class.

initialise master tk class

name = name of current screen’s class

event dict = dictionary of names to a function

if name is mainmenu run screen change to select menu (to pass arguments)

elif name is results screen run screen change to main menu (to pass arguments)

else run the preassigned function

* Init method in header class

initialise master frame class

set style of label prefixed header

set column weight

initialise header text = ttk.label

* Grid frames in header class

grid itself

grid header text

* Init method in intro screen

initialise master frame class

set column weight

set style for label prefixed login

set style for buttons

initialise login frame = tk.frame

set column weight of login frame

initialise login text

initialise username text = ttk.label

initialise username entry = tk.entry

initialise password text = ttk.label

initialise password entry = tk.entry

initialise button frame = tk.frame

initialise register button = ttk.button

initialise login button = ttk.button

* Login method in intro screen

if username entry and password entry

for existing users in users

if username entry != existing user’s username or hashed password != existing user’s password

delete info entered into entry boxes

add user to current users list

configure treeview in main menu

change screen to main menu

else show error

else show error

* Grid frames method in intro screen

grid itself

grid login frame

grid login text

grid username text

grid username entry

grid password text

grid password entry

grid buttons frame

grid register button

grid login button

set focus to username entry

* Init method for the signup class

initialise master frame class

set column weight

set style of label prefixed register

initialise back button = ttk.button

initialise register text = ttk.label

initialise name text = ttk.label

initialise name entry = tk.entry

initialise username text = tkk.label

initialise username entry = tk.entry

initialise password text = ttk.label

initialise password entry = tk.entry

initialise show password button = ttk.button

initialise register button = ttk.button

* Back method for the signup class
* remove all text from login entry widgets

change screen to intro screen

remove all text from the entry widgets

* Register method for the signup class

for existing user in users

if name entry is empty

show error

return

elif username entry or password entry is empty

show error

return

elif username is already taken

show error

return

add user details to database

update users list

show user success message

call back function

* Grid frames method for the signup class

grid back button

grid register button

grid name text

grid name entry

grid username text

grid username entry

grid password text

grid password entry

grid show password button

bind left mouse click on press on show password button

bind left mouse click on lift off on show password button

grid register button

* Init method for the main menu class

initialise master frame class

set style of label prefixed description

set style of label prefixed heading

set column weights

set row weights

set background of itself

initialise frame = tk.frame

set column weights for frame

initialise logout button = ttk.button

initialise description title = ttk.label

initialise description text = ttk.label

initialise user scores title = ttk.label

initialise user scores treeview = ttk.treeview

initialise proceed button = ttk.button

* Treeview configure method for the main menu class

set treeview columns = scores and date

set treeview main heading

set treeview main column

set treeview scores heading

set treeview scores column

set treeview date heading

set treeview date column

* Treeview insert method for the main menu class

delete all data from treeview

update scores list

sort scores list

insert score into treeview for each score in score list

* Static logout method for the main menu class

clear current user list

clear scores list

change screen to intro screen

* Grid frames method for the main menu class

set decription title text for user

call treeview insert method

grid frame

grid logout button

grid description title

grid description text

grid user scores title

grid treeview

grid proceed button

* Init method for the select menu class

Initialise master frame class

set column weights

set row weights

set style of label prefixed title

initialise back button = ttk.button

initialise select text = ttk.label

initialise radio buttons frame = tk.frame

set radio buttons frame’s column’s weight

set radio buttons frame’s row’s weight

initialise var = tk.stringvar

set var as 0

initialise radio buttons list with options

initialise radio buttons = tk.radiobutton for each option in radio buttons list

initialise proceed button = ttk.button

* Proceed method for the select menu class

set question class’ subject to selected radio button

change screen to question

* Grid frames method for the select menu class

grid back button

grid select text

grid radio buttons frame

grid radio button for each radio button in the list

grid proceed button

* Init method for the question class

initialise master frame class

set column weight

set row weight

initialise subject = none

initialise questions list = []

* Frame check method for the question class

if answer entry is empty

show error

return

if questions list == 5

set results class’ user answers list = questions list

set results class’ subject = subject

clear questions list

call get results function from results class

change screen to results screen

return

call grid frames method

* Grid frames method for the question class

if questions list

forget previous questionobj frame

append questionobj frame to questions list

grid newly appended frame

set variable var of newly appended frame = class of standard

grid newly appended frame

* Init method for the questionobj class

initialise master frame class

initialise parent = parent

initialise user answers = none

initialise var dict with index to each standard’s class call

initialise subject according to var dict

set column weight

set row weights

initialise var = tk.stringvar

initialise question = ttk.label

initialise answer entry = ttk.entry

initialise submit button = ttk.button

* Grid frames method for the questionobj class

grid question

grid answer entry

grid submit button

set focus to answer entry

* Init method for the results class

initialise master frame class

initialise user answers = []

initialise subject = none

initialise results = tk.stringvar

set column weight

set row weights

initialise results label = ttk.label

initialise user results = ttk.label

initialise return button = ttk.button

* Get results method for the results class

initialise count = 0

for answer in user answers

var ratio = ratio between subject answer and answer entry

if ratio > ¾

increment count by 1

set results to score out of 5

add score to the database

* Grid frames method for the results class

grid results label

grid user results

grid return button

* Init method for the user class

initialise user id

initialise name

initialise username

initialise password

* Init method for the scores class

initialise user id

initialise date

initialise subject

initialise score

The next methods are in the main routine

* Update users

clear user list

fetch login credentials form database

if fetching is not empty

for user in login creds

append user to user list

* Hash password

return hashed password

* update scores

clear user list

fetch scores from database

if fetching is not empty

for score in user scores

append score to user list

* screen change

forget current screen

current screen = new screen

grid current screen

call grid frames method of current screen

return

The following is for methods in the math operations file

* Init method for surds class

initialise num surds = num surds

initialise surd nums = []

initialise reduced forms = []

initialise sqrt symbol = Unicode for square root symbol

* question gen method for surds class

initialise surd coefficient and number as random integers

append surd coefficient and number to surd nums

initialise closest root = square root of surd number

for factor root in range (closest root, 1, reverse)

initialise factor = factor root squared

if mod (surd number and factor) == 0

append to reduced forms

return

pop numbers from surd nums

call itself

* get answer method for surds class

if reduced form == 1

return sum of coefficients

return sum of coefficients as well as reduced form

* String method for surds class

if surd nums is not empty

return readable question

for I in range(num surds)

call question gen

while reduced forms[i] != reduced forms[i-1]

pop I from surd nums

pop I from reduced forms

call question gen

return readable question

* Init method for complex numbers class

initialise forms list with polar and rectangular

initialise form = none

initialise convert form =- none

initialise equation = none

* Question gen method for complex numbers class

form = random choice between forms

if form == polar

equation list = random number, random number

convert form = other form

return equation

else

equation list = random number, random number

convert form = other form

return equation

* Get answer method for complex numbers class

if form == polar

return alternate form from cmath

else

return alternate form from cmath

* String method for the complex numbers class

call question gen method

return readable question

* Init method for the polynomials class

initialise divisor list with 1

initialise dividend list with 0

initialise empty quotient list

* question gen method for polynomials class

while dividend[0] <= divisor [0]

divisor, dividend = random number

* get answer method for polynomials class

initialise dividends list from dividend list in init

for \_ in range(length(dividend list))

if length(dividend) != length(divisor) – 1

set division = dividend / divisor

append division to quotient

pop first item from dividend list

increment negatively for first item of dividend list

return readable answer

* string method for polynomials class

call question gen function

divisor, dividend = empty list

for list1, list2 in divisor, dividend and divisor, dividend

for index, constant in original list

power = length(original list) – 1

if power – index > 1

append constant x and power to parse list

elif power – index == 1

append constant x to parse list

else append constant to parse list

return readable question

* Init method for the differentiation class

Initialise list with types/different variables to differentiate

Set var to None

set a to None

set x to None

* Question gen method for differentiation class

Set var to random choice from list

Set a and x to random numbers

return formatted question

* Get answer method for differentiation class

If var is 0: return readable answer

elIf var is 1: return readable answer

…

else: return readable answer

* String method for differentiation class

return readable question

* Init method for database class

set con to connection with database

set database cursor

execute a fetch command

set var check to results from fetch

* Initialise method for database class

create users table

create scores table

commit changes

* Fetch method for database class

if login credentials is requested: fetch login credentials

elif scores is requested: fetch scores

return requested data

* Add user method for database class

insert user

commit changes

* Add scores method for database class

insert scores

commit changes

x

### Task 10: Document test cases for testing the program

*Document any testing that can be used to test your program. If any input is inputted using the keyboard, describe the expected input, plus any exceptional, boundary or invalid cases with examples.*

|  |  |  |
| --- | --- | --- |
| **Test cases (Use cases) – Expected, Boundary and Invalid** | **What is expected** | **What actually happened? Screenshot** |
| Invalid | User proceeds with login despite empty input |  |
| Expected | User logs in | Screen changes to main menu |
| Boundary | User enters registration password with exactly 7 characters |  |
| Boundary | User enters registration password with exactly 8 characters |  |
| Boundary | User enters registration password with exactly 9 characters |  |
| Invalid | User proceeds to register with no password |  |
| Invalid | User proceeds to register with no name |  |
| Invalid | User’s desired username is already taken |  |
| Expected | User proceeds onto next question | This is the next question from invalid case |
| Invalid | User proceeds onto next question with blank answer |  |
| Expected | Password turns from \* to text | Note show password button is highlighted as it has been clicked |
| Expected | User’s scores updates after finishing their test |  |
| Expected | Question is generated in a readable format |  |

### Task 11: Refine the plan

*Note any modifications here when iterating through the development cycles.*

## Testing/daily log

|  |  |  |
| --- | --- | --- |
| Date | **What was tested** | **Screen shot and how have you fixed** |
| **29/7** | Login function would not work for any accounts except for first one | Changed the conditional to show an error if the login fails first instead of showing the error for the first iterated attempt |
| **29/7** | Proportions of main menu screen where instructions text pushed rest of treeview details out | Played around with different text sizes until it fit perfectly. Works as the instructions text will not change |
| **30/7** | Questions where user would obtain 0 score despite correct answers | Typo in the conditional ratio of string relativity from 4/3 (which is impossible) to ¾ |
| **30/7** | Show password button would not reveal the text | Onclick and release of the button were incorrectly binded due to the lack of use of lambda to pass the correct form the *show* argument should be |
| **27/10** | The original program did not show immediate feedback on whether the answer was correct or incorrect. It would also not show the expected answer leading to the users not knowing which questions were wrong and how they were wrong | An introduction of immediate feedback system that displays whether the user was correct or incorrect, and the answer if the user was incorrect so the user can compare the answers themselves. |
| **27/10** | Name space errors | To avoid namespace errors from popping up in the code, we import the module as is so when we use the module, we have to prefix it with the module’s name. For example, all tkinter and tkinter.ttk commands must be prefixed with tk and ttk respectively. |
| **27/10** | Organisation of Tk() as it is the root of the window and configuration of it has to be ‘neater’. | All configurations of the root class are done in the Tk(), as well as a static method for the button click events. Arguably, the same thing could be achieved without a class and without messing up the code, hence implemented directly into the main routine. However, this is up to personal preference and doesn’t affect much. |
| **27/10** | Organisation of each frame within the program into an object-oriented approach. | The major portions of the program such as the header and each screen are all split into different classes. This object-oriented approach makes organisation a lot easier and as each class becomes an instance, resetting the frame as well as calling each other becomes easier with a controller as well. This is an example of a MVC model or model-view-controller which is a software design pattern commonly used for UI. The way it follows the MVC model is that the class itself a frame, becoming the ‘view’ component, the class methods are the controller as the user interacts with it and calls its methods, and the parent methods are the model aspect where it manipulates the backend and at times, the frontend. This also gives a private namespace for each variable in the class due to the nature of classes. |
| **27/10** | Splitting the major components of the program into different files for organisation purposes. | The main program has been split into three different files for each major portion of the program. First we have the main class with deals with the tkinter side and hence the front end. Next is the math operations file which deals with the mathematical side such as creation of the question and answer. This is part of the backend. There is also the database file which deals with queries to the database. This is also a part of the backend. Splitting off these modules into different files makes it a lot easier to conduct testing as well as incremental development in improving the efficiency of the code. |
| **27/10** | Using \_\_name\_\_ over \_\_repr\_\_ in the classes in math operations and instead of assigning the question and answer a variable | The use of \_\_name\_\_ sets the string of the class, when called, to execute the following commands and then return that string. This makes it so when something asks for the string value such as the text argument in tkinter’s label, it is much easier and more efficient to simply enter in the class reference rather than the class reference and the variable name. This also makes it so the methods used in the \_\_name\_\_ function does not execute upon class initialisation and only when the string method is called, making runtime slightly more efficient. |
| **27/10** | Using instance classes to its fullest | This is the question class, or more specifically the object part of the question class. This class will be called every time a question is requested and will generate the widgets that the user enters. This makes it so that there is no need to use multi-dimensional list or other data structures to store each response and answer, only the class needed to be stored in a list and could later, have its own methods called from other classes, following the MVC model very closely. With the later addition of the instant feedback system, this does not become as significant but is still optimal in my opinion as there is no need for any label changes and such, which would feel very unorganised otherwise. |
| **27/10** | Support class for uses and scores | The implementation of a support class for storing the users and the scores is good here since it removes the need for a multi-dimensional list or parallel list. For example, in the users list, there are 4 variables that need to be stored, 3 as user id can be removed due to the iteration and how user id is generated. This will still require a 2d list or 3 parallel lists which is hard to iterate through and indexing makes the debugging process harder as the developer has to constantly refer back to see which index refers to what. |
|  |  |  |
|  |  |  |

# Assessment schedule: ICC T20 World Cup Ticketing System

| Evidence/Judgements for  Achievement | Evidence/Judgements for Achievement with Merit | Evidence/Judgments for Achievement with Excellence |
| --- | --- | --- |
| The student has developed a complex computer program for the specified task.  The student has designed and implemented a program which:  · met most of the specifications outlined in the task for “CWG-GC2018 Ticketing”  · included variables, an indexed data structure, and a modular structure including details of the procedural structures of the modules  · included a working graphical user interface with different sources of event generating components and event handling, and used classes and objects to encapsulate data and methods  · set out the program code clearly and documented the program with comments  · tested and debugged the program to ensure it worked on a sample of expected input cases    The student has completed a working program to meet specifications using a documented design process, but may:  · not have anticipated user errors, for example, multiple decimal points within one number  · have included redundant pieces of code  · have poorly named variables, for example, x instead of current\_input  · not be able to articulate what specific pieces of code are doing, having developed them by trial and error with little understanding  · have failed to properly contain all aspects of the logic or state of the stock monitor within the support class.  · included a working graphical user interface with different sources of events generating components and event handling, and used classes and objects to encapsulate data and methods  The student has included a functioning user interface, but it may:  · respond to a button click and the corresponding keyboard event with duplicated and repeated code instead of the same code  · have individually named buttons and then stored them in an indexed data structure as well, only because the criteria requires it  · distinguish between events with awkward/ inefficient if statements instead of, for example, if/elif/else  · respond to number inputs individually instead of as a group of possible inputs requiring the same response  · respond to operator inputs individually instead of as a group of possible inputs requiring the same response.  Set out the program code clearly and document the program with comments  · For example, a comment for code that clears all values out ready for next release of tickets: “Clears out and resets.”  Tested and debugged the program to ensure it works on a sample of expected input cases  · The program works on clean input such as clicking the appropriate buttons to Sell or Reset the shows  This description relates to only part of what is required, and is indicative only. | The student has skilfully developed a complex computer program for the specified task.  The student has designed and implemented a program which:  · met all the specifications outlined in the task for “CWG-GC2018 Ticketing”  · used well-chosen modular and procedural structures, scope and encapsulation for data and methods, graphical user interface and event handling mechanisms  · documented the program with variable and module names and comments that accurately describe code function and behaviour  · followed a disciplined design and implementation process, documenting cycles of incremental development and documenting and conducting comprehensive testing processes, to ensure that the program worked on inputs that included both expected and boundary cases    The student has completed a working program to meet specifications, following a disciplined design, implementation, and comprehensive testing process with documented cycles of incremental development  The student has, for example:  · anticipated and responded to user errors, for example, preventing the use of a string in the sellingfunction, but perhaps only in such a way that the user must restart the whole restocking process  · thoughtfully edited their code, removing redundancy  · understood the purpose of all the code they have written and are able to articulate that purpose.  Has used well-chosen scope and encapsulation for data, control structures, elements of the programing language, and event handling mechanisms  The student has, for example:  · mostly maintained a clear distinction between the GUI class and the logic/support class  · created and added buttons to a list(s) using a loop(s) and a list(s) of strings  · bound the handler(s) to keyboard events using the same technique  · placed buttons using loop(s)  · responded to a button click and the corresponding keyboard event with the same code  · responded to logical groups of possible user inputs (for example, all numbers) with the same code  · ensured that a value that is only ever required in one module should not belong to the whole class.  Documented the program with variable and module names and comments that accurately describe code function and behaviour.  · The student has named variables descriptively, for example, current\_input, current\_operator etc.  · The comments used accurately describes code    This description relates to only part of what is required, and is indicative only. | The student has efficiently developed a complex computer program for the specified task.  The student has designed and implemented a program:  · in which the overall modular and procedural design, graphical user interface, and event handling design, were a well-structured, logical decomposition of the task, and the resulting program is flexible and robust  · which set out the program code concisely and documented the program with comments that explained and justified decisions  · where there was evidence of comprehensive testing and debugging in an organised and time effective way and which ensured the program was correct on expected, boundary and invalid input cases        The student maintained a clear distinction between the GUI class and the logic/support class so that the logic class is easily reusable for a differently styled GUI or a different context, for example, command line  · designed the GUI to be easily adapted, for example, if buttons for adding another stock item were suddenly required to be added the extra code needed would be minimal  · designed the handler to be easily adapted, for example, if functionality for adding another stock item were suddenly required the extra code needed would be minimal  · anticipated user errors (for example, multiple clicks, strings where numbers were expected or leading zeroes) and responded gracefully, for instance, excluded ‘impossible’ entries and given the user the chance to recover without losing ‘work’ or simply being told ‘ERROR’  Set out the program code concisely and documented the program with comments that explain and justify decisions    This description relates to only part of what is required, and is indicative only. |

Final grades will be decided using professional judgement based on a holistic examination of the evidence provided against the criteria in the Achievement Standard