DBMS Project Report

1. Problem Statement:

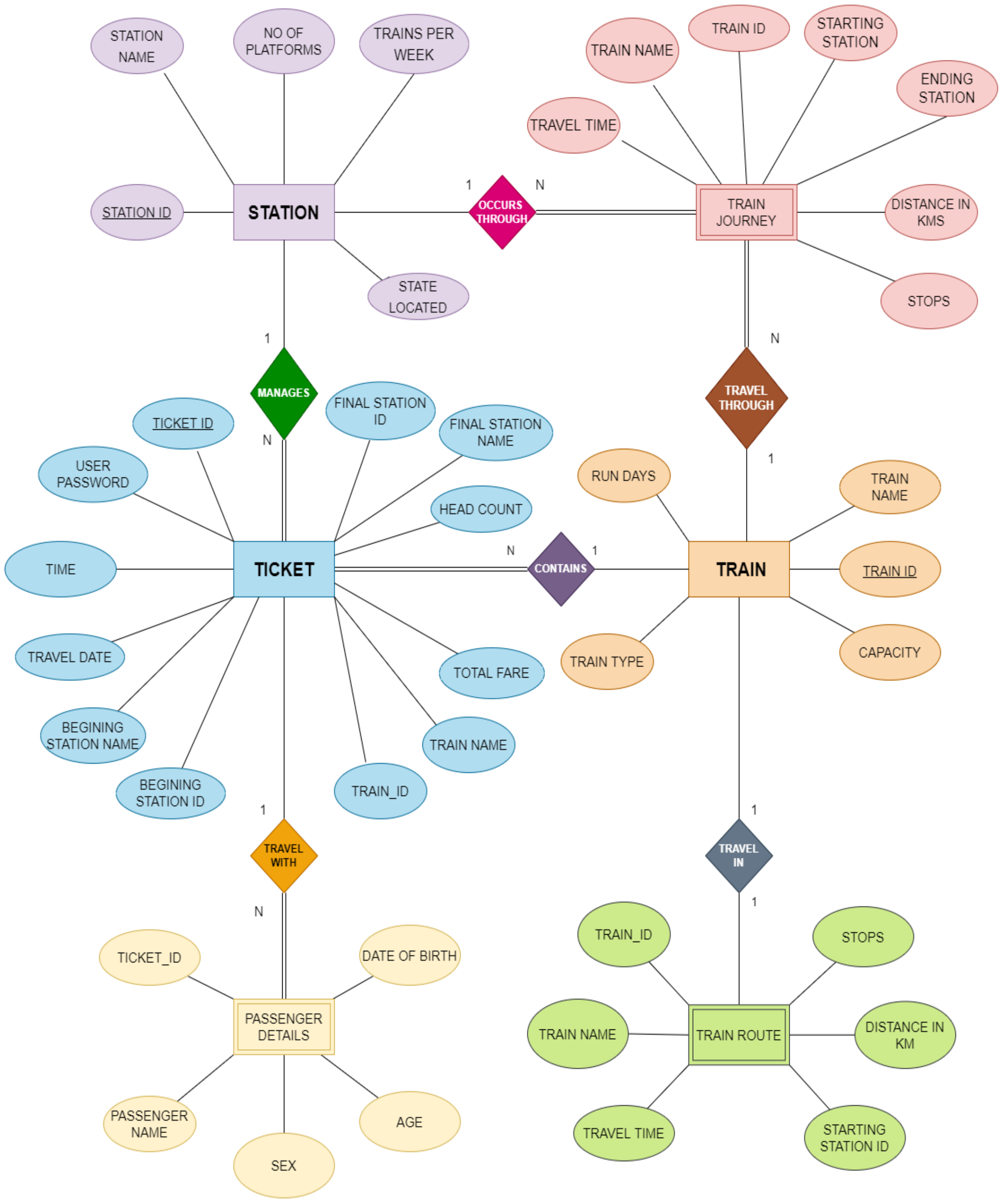
Railway Enquiry System

Design a database solution for the railway network. The enquiry system should have the following information:

- Station names
- Train IDs with names
- Weekly Schedules of the trains
- Ticket Availability
- Cost of trip

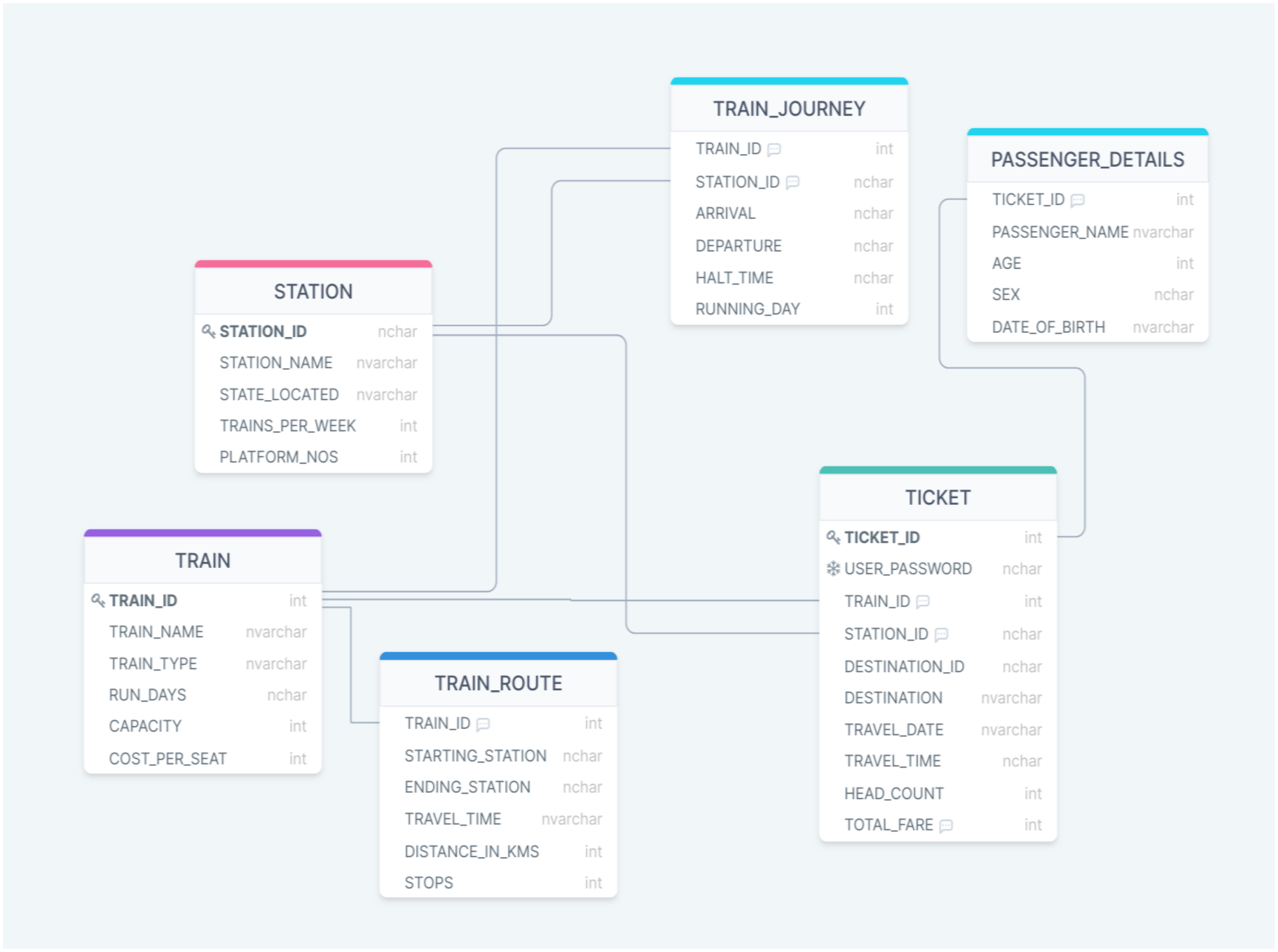
The train schedules should have information on the stations from where the train starts and by when it reaches the destination. It should also include information about all stations it passes through during its journey.

2. ER diagram.


STATION NAME
NO OF PLATFORMS
TRAINS PER WEEK
STATION ID
STATION
STATE LOCATED
1
OCCURS THROUGH
N
TRAIN NAME
TRAIN ID
STARTING STATION
ENDING STATION
TRAVEL TIME
TRAIN JOURNEY
DISTANCE IN KMS
STOPS
1
MANAGES
N
N
TRAVEL THROUGH
1
TICKET ID
FINAL STATION ID
FINAL STATION NAME
USER PASSWORD
HEAD COUNT
TIME
TICKET
N
CONTAINS
1
RUN DAYS
TRAIN NAME
TRAIN
TRAIN ID
TRAIN TYPE
CAPACITY
TRAVEL DATE
TOTAL FARE
BEGINING STATION NAME
TRAIN NAME
BEGINING STATION ID
TRAIN_ID
1
TRAVEL WITH
N
1
TRAVEL IN
1
TICKET_ID
DATE OF BIRTH
PASSENGER DETAILS
PASSENGER NAME
SEX
AGE
TRAIN_ID
STOPS
TRAIN NAME
TRAIN ROUTE
DISTANCE IN KM
TRAVEL TIME
STARTING STATION ID

3. Database schema as required for building the application.

1. STATION(<u>Station_id</u>,Station_name,State_located,Trans_per_week,Platform_nos)
2. TRAIN(<u>Train_id</u>,Train_name,Train_type,Run_days,Capacity,Cost_per_seat)
3. TRAIN_ROUTE(Train_id,Starting_station,Ending_station,Travel_time,Distance_in_kms ,Stops)
4. TRAIN_JOURNEY(Train_id, Station_id, Arrival,Departure,Halt_time,Running_day)
5. TICKET(Ticket_id,User_password,Train_id, Station_id, Arrival,Departure,Halt_time,Running_days)
6. PASSENGER_DETAILS(Ticket_id,Passenger_name,Age,Sex,Date_of_birth)



4.Database normalization and justification.

A table is required to store the list of all the teams and their managers with their respective details.
List_of_teams (TeamName, Stadium, ManagerName, ManagerAge, ManagerStyle)
The above table is clearly satisfying 1NF since all the column name are unique and all the columns would contain atomic values and all the data present in a column are of the same type.
But a deletion anomaly occurs when we try to delete all the records from the table.
When we delete all the team details (or records) from the table, we accidentally delete all the manager information as well. So, to avoid this kind of anomaly we split the table into two different tables i.e.
ManagerDetails (M_id, ManagerName, Age, Style)
List_of_Teams (TeamName, M_id, Stadium)
Here in the above tables, M_id is a primary key in the ManagerDetails table and M_id is a foreign key in the List_of_Teams table. When the table is split into two tables and now when we delete all the team details (or records) from the List_of_Teams table, the manager details remain unaltered. So, we can say that our tables now satisfy the 2NF form since it is devoid of deletion anomaly that we previously encountered and there are no partial dependencies.
Our Tables (List_of_teams & ManagerDetails) are also satisfying the 3NF form since there is no transitive dependencies.

Another table is required to store the details of all the player of each team.
List_of_Players (Player_id {TeamName, Jersey No}, Name, Pos, DOB, AGE)
Here TeamName and Jersey No of the player together forms a composite key which and which uniquely identifies all the records of our List_of_Players table.
The above table (List_of_Players) is clearly satisfies 1NF since all the column name are unique and all the columns would contain atomic values and all the data present in a column are of the same type.
Here in this table TeamName is a foreign key which refers to the TeamName in the List_of_Teams table.
The List_of_Players table is also satisfying the 2NF and 3NF since there are no partial and transitive dependencies. And the redundant data is minimized and it is devoid of all the anomalies.

A table is required to store all the matches that took place between different teams and all the players who played in a particular match.

We could create a single table that contains all the details of the players who played in different matches but doing so would lead to a lot of redundant data in our table and can cause insertion, deletion and updation anomalies.
So, we split the table we intended to create into 3 different tables namely
Fixtures (F_id, Date, time, HomeTeam, AwayTeam, score)
Homeplayers (F_id, HomeTeam, Jersey_No, Playing11)
Awayplayers (F_id, Awayteam, Jersey_No, Playing11)
The fixtures table would contain all the previous match information like the date, time, score etc. whereas the Homeplayers and the Awayplayers table would contain whether a particular player was in the match as one of the playing 11 players or if he was in the substitutes or reserves.
The F_id is the primary key in the fixtures table and is a foreign key in the tables Homeplayers and Awayplayers.
The fixtures table satisfies the 1NF, 2NF and the 3NF since all the column names are unique and the values are atomic and it doesn't have any partial or transitive dependencies.
In the Homeplayers table, we can identify each record of the table uniquely by considering the F_id, HomeTeam and Jersey_no together as a primary key. So therefore (F_id, HomeTeam, Jersey_no) together forms the composite key which can uniquely identify each record of the table.
The Homeplayers table satisfies the 1NF form since all the column names are unique and the values are atomic. It also satisfies the 2NF form since it doesn't have any partial dependencies because of the functional dependency
{F_id, HomeTeam, Jersey_no} → Playing11 i.e. all the 3 columns are required to identify if a player was in the Playing11 or not.
The Homeplayers tables also satisfies the 3NF form since there aren't any transitive dependencies because no non-prime attribute depends on any other non-prime attribute.

Similarly, we can say the same for the Awayplayers table too.

Therefore, all the tables in the databased are normalized to remove all the anomalies and to reduce redundant data.

5. Coding & Implementation:

```
import sqlite3
from tkinter import *
from tkinter import ttk
import tkinter as tk
from tkinter import messagebox as msg
from PIL import Image, ImageTk


#To set connection with sqlite3 Railway.db

def connection():
    con=sqlite3.connect('Railway.db')
    cursor=con.cursor()
    return con,cursor


#Login function gets triggred on pressing LOGIN button

def login():
    if not e_u.get().isnumeric():
        e_u.set("")
        e_p.set("")
        msg.showerror('ERROR','Incorrect Userid/Password!')
    else:
        id=int(e_u.get())
        pwd=e_p.get()
        e_u.set("")
        e_p.set("")
        con,cur=connection()
        cur.execute("select * from ticket where ticket_id={} and 
user_password='{}'".format(id,pwd))
        result=cur.fetchall()
        if result!=[]:
            for i in result:
                ticket_id=i[0]
                if id!='' and pwd!='':
                    msg.showinfo("",'Login sucessful')
                    login_screen.destroy()
                    personal_details(ticket_id)
        else:
            msg.showerror('ERROR','Incorrect Userid/Password!')
    con.close()
```

```
#window to display Personal details of the passenger based on the Ticket ID

def personal_details(ticket_id):
    info = Tk()
    info.geometry('1170x300')
    info.title('Details')
    info.iconbitmap('Logo.ico')
    title_label = Label(info,text="Ticket Infomation",font=("Calibri bold", 18))
    title_label.pack(pady=10)

    table_frame=Frame(info).pack(padx=50,pady=20)
    tree=ttk.Treeview(table_frame,height=3)

    tree["columns"]=("1","2","3","4","5","6","7","8","9","10")

    tree.column("#0", width=120, minwidth=110, stretch=NO,anchor=CENTER)
    tree.column("1", width=50, minwidth=50, stretch=NO,anchor=CENTER)
    tree.column("2", width=50, minwidth=50,stretch=NO,anchor=CENTER)
    tree.column("3", width=130, minwidth=100, stretch=NO,anchor=CENTER)
    tree.column("4", width=210, minwidth=170, stretch=NO,anchor=CENTER)
    tree.column("5", width=160, minwidth=100, stretch=NO,anchor=CENTER)
    tree.column("6", width=150, minwidth=100, stretch=NO,anchor=CENTER)
    tree.column("7", width=100, minwidth=80, stretch=NO,anchor=CENTER)
    tree.column("8", width=100, minwidth=80, stretch=NO,anchor=CENTER)
    tree.column("9", width=100, minwidth=80, stretch=NO,anchor=CENTER)

    tree.heading("#0",text="PASSENGER_NAME")
    tree.heading("1", text="AGE")
    tree.heading("2", text="SEX")
    tree.heading("3", text="DATE_OF_BIRTH")
    tree.heading("4", text="TRAIN_NAME")
    tree.heading("5", text="ORIGIN")
    tree.heading("6", text="DESTINATION")
    tree.heading("7", text="TRAVEL_DATE")
    tree.heading("8", text="DEPARTURE_TIME")
    tree.heading("9", text="FARE_PER_HEAD")

    tree.pack(fill=X)

    btnFrame = Frame(info)
    schedule_button=ttk.Button(btnFrame,text="     TICKET SCHEDULE
 ",takefocus=False, command=lambda: schedule(info))
```

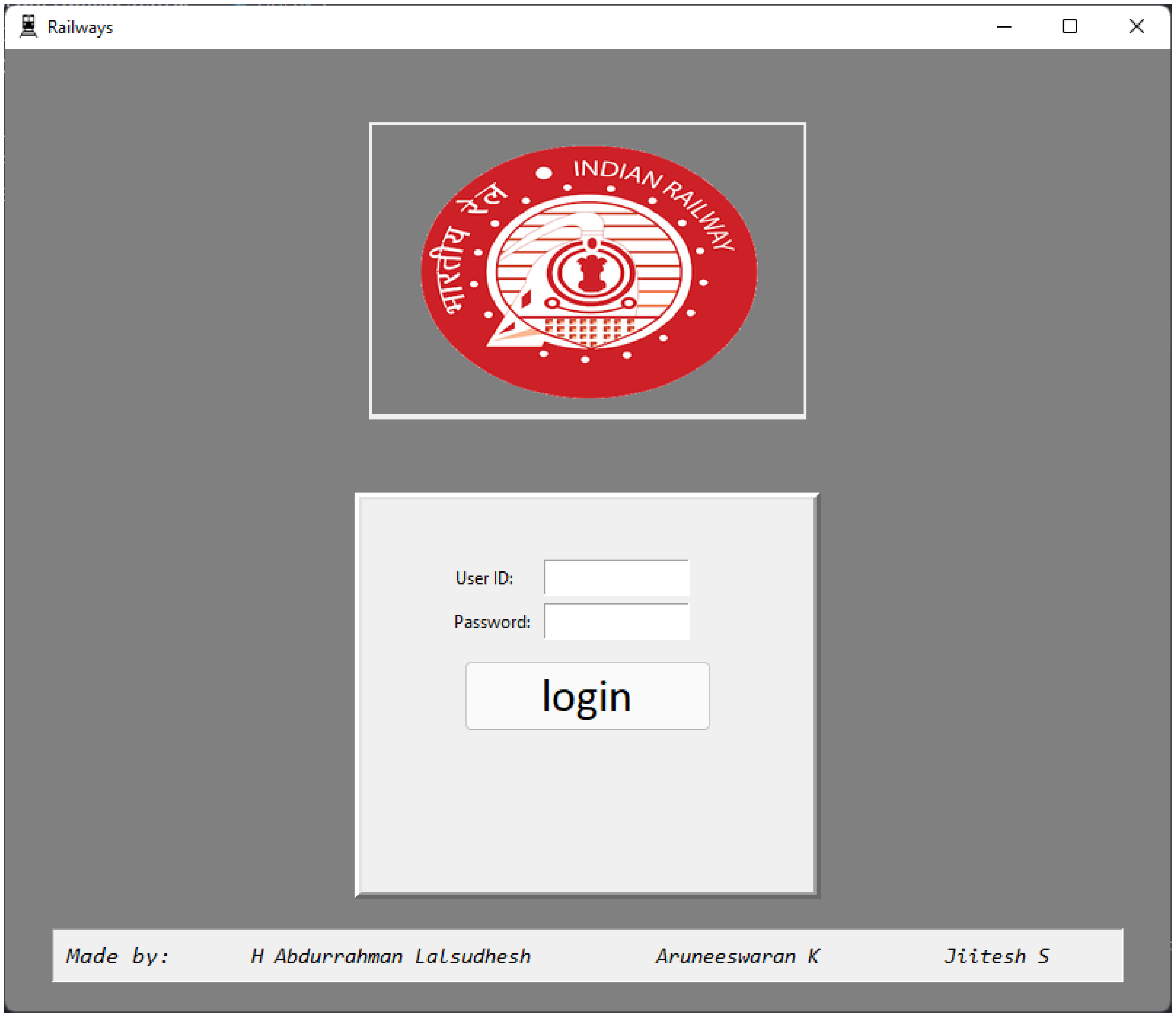
```
    schedule_button.pack(pady=30,side=BOTTOM)
    btnFrame.pack(fill=X)

    # inserting values to tree
    con, cur = connection()
    lst = cur.execute("select
PASSENGER_NAME,AGE,SEX,DATE_OF_BIRTH,TRAIN_NAME,STATION_NAME,DESTINATION,TRAVEL_D
ATE,TRAVEL_TIME,TOTAL_FARE,HEAD_COUNT from PASSENGER_DETAILS natural join TICKET
where TICKET_ID={}".format(ticket_id))

    for tup in lst:
        d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,d10=tup
        tree.insert("", "end", text=str(d0), values=(str(d1), str(d2), str(d3),
str(d4), str(d5),str(d6),str(d7),str(d8),str(d9/d10)))

    tk.Label(info,text='Cost of Trip:
 {}'.format(d9),relief=RIDGE).place(width=150,height=30,x=1000,y=200)
    con.close()




#schedule window to display the weekly schedule of the Trains

def schedule(win):
    win.destroy()
    schedule = Tk()
    schedule.geometry('870x400')
    schedule.title('Ticket Schedule')
    schedule.iconbitmap('Logo.ico')
    title_label = Label(schedule,text='Weekly Schedule',font=("Calibri bold",
17))
    title_label.pack(pady=10)

    table_frame = Frame(schedule)
    table_frame.pack()
    sb = ttk.Scrollbar(table_frame)
    tree=ttk.Treeview(table_frame, height=9,yscrollcommand = sb.set)
    tree["columns"]=("1","2","3","4","5","6")

    tree.column("1", width=90, minwidth=100, stretch=NO,anchor=CENTER)
    tree.column("#0", width=190, minwidth=120, stretch=NO,anchor=CENTER)
    tree.column("2", width=140, minwidth=100,stretch=NO,anchor=CENTER)
    tree.column("3", width=130, minwidth=100,stretch=NO,anchor=CENTER)
    tree.column("4", width=100, minwidth=100,stretch=NO,anchor=CENTER)
    tree.column("5", width=100, minwidth=100,stretch=NO,anchor=CENTER)
    tree.column("6", width=100, minwidth=100,stretch=NO,anchor=CENTER)

    tree.heading("#0",text="TRAIN_NAME")
    tree.heading("1", text="TRAIN_ID")
    tree.heading("2", text="STARTING_STATION")
    tree.heading("3", text="ENDING_STATION")
    tree.heading("4", text="TRAVEL_TIME")
    tree.heading("5", text="DISTANCE_IN_KMS")
    tree.heading("6", text="STOPS_NOS")

    sb.config(command = tree.yview)
    sb.pack(side = RIGHT, fill=Y)
    tree.pack(fill=X)

    btnFrame = Frame(schedule)
    ttk.Button(btnFrame,text="STATIONS PASSING",takefocus=False, command=lambda:
station_details(tree)).pack(pady=8)#height=50,width=120,x=100,y=320
    ttk.Button(btnFrame,text="TRAIN DESCRIPTION",takefocus=False,command=lambda:
train_desc(tree)).pack(pady=8)#side=BOTTOM
    ttk.Button(btnFrame,text="TICKET
AVAILABILITY",takefocus=False,command=lambda:
ticket_availability(tree)).pack(pady=8)#padx=20,pady=10,side=RIGHT
    btnFrame.pack(fill=X)

    # inserting values to tree
    con, cur = connection()
    lst = cur.execute("select
TRAIN_NAME,TRAIN_ID,STARTING_STATION,ENDING_STATION,TRAVEL_TIME,DISTANCE_IN_KMS,S
TOPS from TRAIN_ROUTE")
    for tup in lst:
            d0,d1,d2,d3,d4,d5,d6=tup
            tree.insert("", "end", text=d0, values=(str(d1), str(d2), str(d3),
str(d4), str(d5),str(d6)))
    con.commit()
    con.close()
```

```
#Station_details to display the stations a selected train passes through based on
it ID

def station_details(tree):
    train_id=tree.item(tree.selection())['values'][0]
    stations=Toplevel()
    stations.geometry('700x300')
    stations.iconbitmap('Logo.ico')
    stations.title('Stations')
    stations.config(bg='#4DC1A2')
    con,curr = connection()
    curr.execute('SELECT TRAIN_NAME FROM TRAIN WHERE
TRAIN_ID={}'.format(train_id))
    value=curr.fetchall()
    title_label = Label(stations,text='{} PASSES THROUGH THE
STATIONS'.format(value[0][0]),font=("Calibri bold", 17),bg='#4DC1A2')
    title_label.pack(pady=10)
    curr.close()

    table_frame = Frame(stations)
    table_frame.pack()

    sb = ttk.Scrollbar(table_frame)
    tree=ttk.Treeview(table_frame, height=7,yscrollcommand = sb.set)
    tree["columns"]=("1","2","3","4")

    tree.column("#0", width=190, minwidth=120, stretch=NO,anchor=CENTER)
    tree.column("1", width=90, minwidth=100, stretch=NO,anchor=CENTER)
    tree.column("2", width=140, minwidth=100,stretch=NO,anchor=CENTER)
    tree.column("3", width=130, minwidth=100,stretch=NO,anchor=CENTER)
    tree.column("4", width=100, minwidth=100,stretch=NO,anchor=CENTER)

    tree.heading("#0",text="STATION_NAME")
    tree.heading("1", text="ARRIVAL")
    tree.heading("2", text="DEPARTURE")
    tree.heading("3", text="HALT_TIME")
    tree.heading("4", text="RUNNING_DAY")

    sb.config(command = tree.yview)
    sb.pack(side = RIGHT, fill=Y)
    tree.pack(fill=X)

    con, cur = connection()
    lst = cur.execute("select
STATION_NAME,ARRIVAL,DEPARTURE,HALT_TIME,RUNNING_DAY from TRAIN_JOURNEY where
TRAIN_ID={} order by RUNNING_DAY,departure".format(train_id))
    for tup in lst:
            d0,d1,d2,d3,d4=tup
            tree.insert("", "end", text=d0, values=(str(d1), str(d2), str(d3),
str(d4)))

    con.commit()
    con.close()


#Train_desc to display the description of a selected train based on its ID

def train_desc(tree):
    train_id=tree.item(tree.selection())['values'][0]
    description=Toplevel()
    description.geometry('750x150')
    description.iconbitmap('Logo.ico')
    description.title('Stations')
    description.config(bg='#9F94BF')
    con,curr = connection()
    curr.execute('SELECT TRAIN_NAME FROM TRAIN WHERE
TRAIN_ID={}'.format(train_id))
    value=curr.fetchall()
    title_label = Label(description,text='DESCRIPTION OF
{}'.format(value[0][0]),font=("Calibri bold", 17),bg='#9F94BF')
    title_label.pack(pady=10)
    curr.close()

    table_frame = Frame(description)
    table_frame.pack()

    tree=ttk.Treeview(table_frame, height=1)
    tree["columns"]=("1","2","3","4")

    tree.column("#0", width=190, minwidth=140, stretch=NO,anchor=CENTER)
    tree.column("1", width=150, minwidth=100, stretch=NO,anchor=CENTER)
    tree.column("2", width=140, minwidth=100,stretch=NO,anchor=CENTER)
    tree.column("3", width=130, minwidth=100,stretch=NO,anchor=CENTER)
    tree.column("4", width=100, minwidth=100,stretch=NO,anchor=CENTER)

    tree.heading("#0",text="TRAIN_NAME")
    tree.heading("1", text="TRAIN_TYPE")
    tree.heading("2", text="RUN_DAYS")
    tree.heading("3", text="TOTAL_CAPACITY")
    tree.heading("4", text="COST_PER_SEAT")
    tree.pack(fill=X)

    con, cur = connection()
    lst = cur.execute("select
TRAIN_NAME,TRAIN_TYPE,RUN_DAYS,CAPACITY,COST_PER_SEAT from train where
TRAIN_ID={}".format(train_id))
    for tup in lst:
            d0,d1,d2,d3,d4=tup
            tree.insert("", "end", text=d0, values=(str(d1), str(d2), str(d3),
str(d4)))

    con.commit()
    con.close()


#ticket_availability to display the availability of tickets in a selected train

def ticket_availability(tree):
    train_id=tree.item(tree.selection())['values'][0]
    con,cur=connection()
    cur.execute('select capacity-(select sum(head_count) as total_passengers from
ticket group by train_id having train_id={}) from train where
train_id={}'.format(train_id,train_id))
    res=cur.fetchall()
    msg.showinfo('Ticket availability','Available tickets\n
 {}'.format(res[0][0]))


#LOGIN SCREEN
#__main__:
login_screen=Tk()
login_screen.geometry('800x660')
login_screen.title('Railways')
login_screen.iconbitmap('Logo.ico')
login_screen.configure(bg='grey')
s = ttk.Style()
s.configure('TButton', font=('Calibri', 24))
s.configure("Treeview.Heading", font=('Calibri', 14))
s.configure('Treeview', rowheight=27, font=('Calibri', 12))


img= (Image.open("IRCTC.png"))
resized_image= img.resize((300,200), Image.ANTIALIAS)
```

```
new_image= ImageTk.PhotoImage(resized_image)
canvas= Canvas(login_screen, width= 500, height= 200)
canvas.create_image(0,0, anchor=NW, image=new_image)
canvas.pack(padx = 250, pady= 50)

f1=Frame(login_screen,bd=5,relief=RAISED) #must be flat, groove, raised, ridge,
solid, or sunken
f1.pack()

f2=Frame(login_screen,bd=1,relief=SUNKEN)
f2.pack(fill=X,padx = 32, pady = 20)
names_frame = Frame(f2)
names_frame.pack(pady = 10)
Label(names_frame,text="Made by: ",font=("Consolas",12,'italic')).pack(side=LEFT)
Label(names_frame,text="H Abdurrahman
Lalsudhesh",font=("Consolas",11,'italic')).pack(side=LEFT, padx= 40)
Label(names_frame,text="Aruneeswaran
K",font=("Consolas",11,'italic')).pack(side=LEFT, padx = 40)
Label(names_frame,text="Jiitesh S",font=("Consolas",11,'italic')).pack(side=LEFT,
padx = 40)

label_userid=tk.Label(login_screen,text='User
ID:',justify=LEFT).place(width=70,height=25,x=294,y=350)
label_password=tk.Label(login_screen,text='Password:').place(width=70,height=25,x
=300,y=380)

e_u=tk.StringVar()
e_p=tk.StringVar()
entry_userid=tk.Entry(login_screen,textvariable=e_u).place(width=100,height=25,x=
370,y=350)
entry_password=tk.Entry(login_screen,show='*',textvariable=e_p).place(width=100,h
eight=25,x=370,y=380)

border2 = Frame(f1)
login_button=ttk.Button(border2,text="login",style="W.TButton",width=10,takefocus
=False, command=login)
login_button.pack(pady=20,side=BOTTOM)
border2.pack(padx=70,pady=90)

login_screen.mainloop()
```

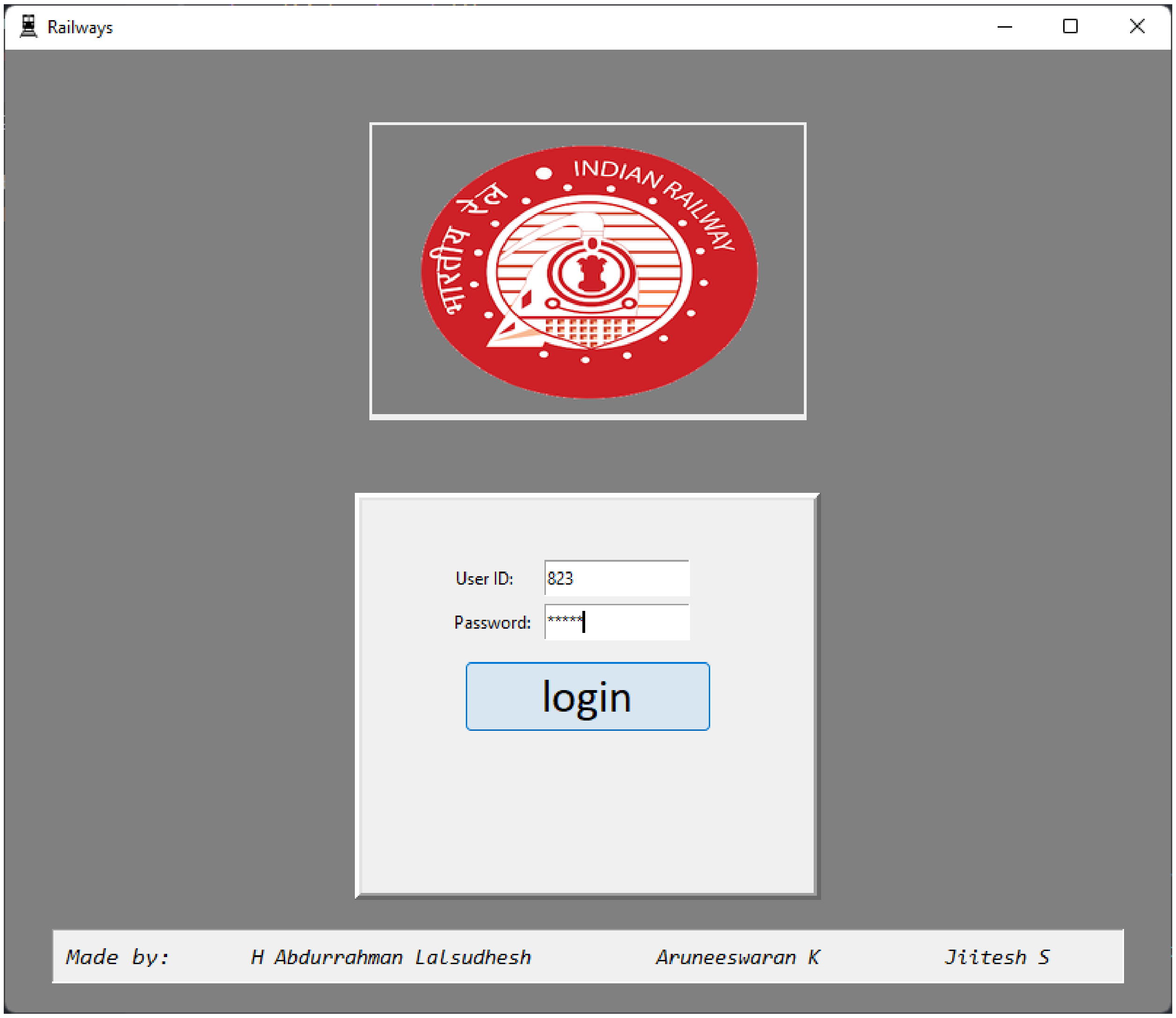
6. Results (Screenshots).

Fig 1. Login screen window.



Made by: H Abdurrahman Lalsudhesh Aruneeswaran K Jiitesh S

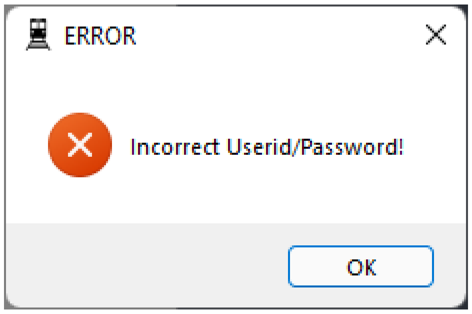
Fig 2. Login screen with error.



User ID: 823

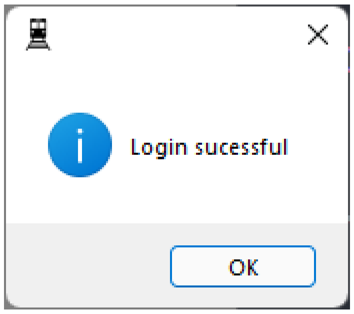
Password: *****

Made by: H Abdurrahman Lalsudhesh Aruneeswaran K Jiitesh S



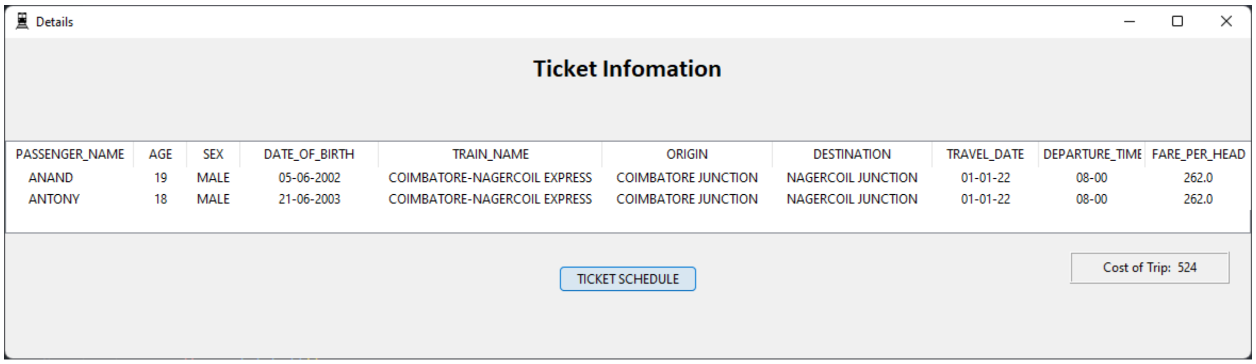
Incorrect Userid/Password!

Fig 3. Successfully logged in screen.



Login sucessful

Fig 4. Ticket information window.



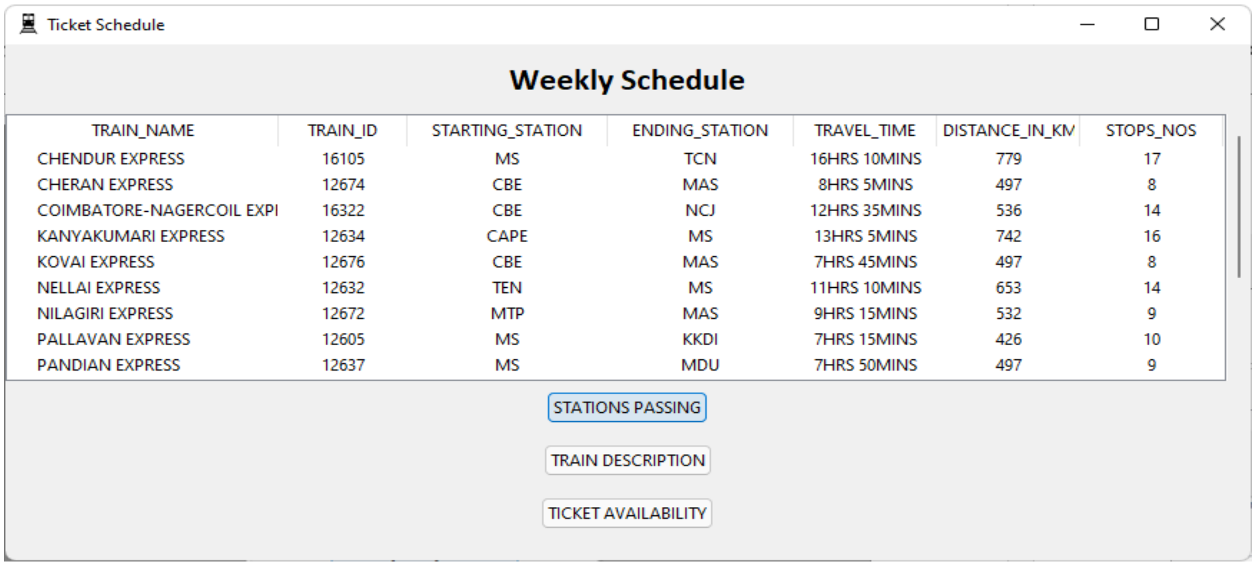
Details

Ticket Infomation

PASSENGER_NAME	AGE	SEX	DATE_OF_BIRTH	TRAIN_NAME	ORIGIN	DESTINATION	TRAVEL_DATE	DEPARTURE_TIME	FARE_PER_HEAD
ANAND	19	MALE	05-06-2002	COIMBATORE-NAGERCOIL EXPRESS	COIMBATORE JUNCTION	NAGERCOIL JUNCTION	01-01-22	08-00	262.0
ANTONY	18	MALE	21-06-2003	COIMBATORE-NAGERCOIL EXPRESS	COIMBATORE JUNCTION	NAGERCOIL JUNCTION	01-01-22	08-00	262.0

Cost of Trip: 524

Fig 5. Ticket schedule window

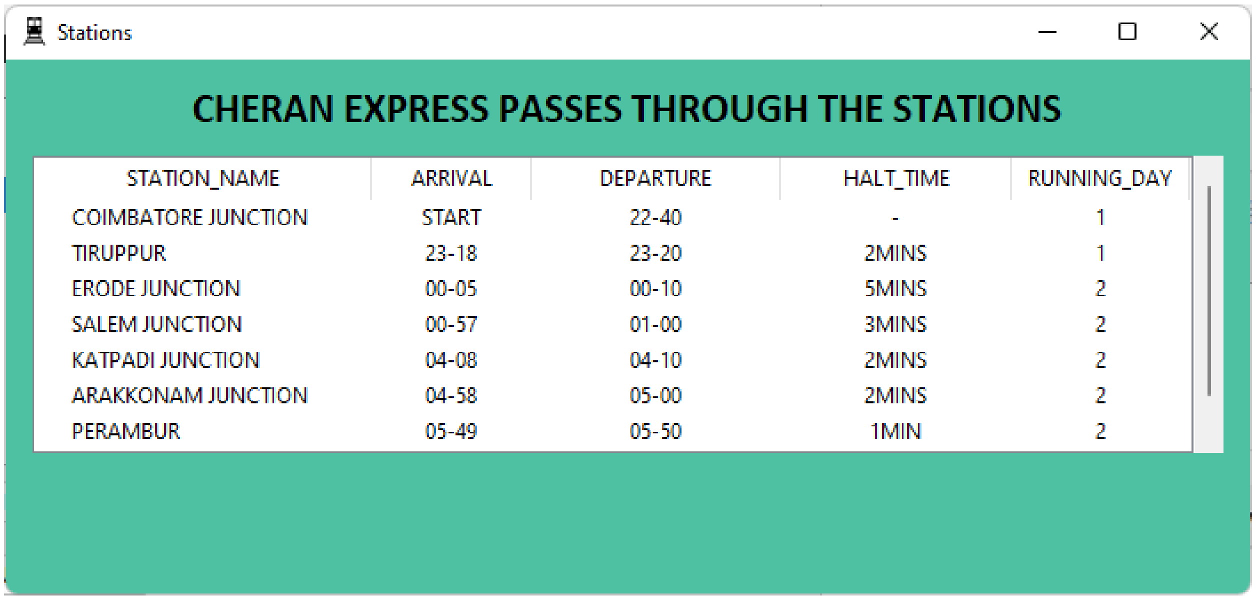


Ticket Schedule

Weekly Schedule

TRAIN_NAME	TRAIN_ID	STARTING_STATION	ENDING_STATION	TRAVEL_TIME	DISTANCE_IN_KM	STOPS_NOS
CHENDUR EXPRESS	16105	MS	TCN	16HRS 10MINS	779	17
CHERAN EXPRESS	12674	CBE	MAS	8HRS 5MINS	497	8
COIMBATORE-NAGERCOIL EXPI	16322	CBE	NCJ	12HRS 35MINS	536	14
KANYAKUMARI EXPRESS	12634	CAPE	MS	13HRS 5MINS	742	16
KOVAI EXPRESS	12676	CBE	MAS	7HRS 45MINS	497	8
NELLAI EXPRESS	12632	TEN	MS	11HRS 10MINS	653	14
NILAGIRI EXPRESS	12672	MTP	MAS	9HRS 15MINS	532	9
PALLAVAN EXPRESS	12605	MS	KKDI	7HRS 15MINS	426	10
PANDIAN EXPRESS	12637	MS	MDU	7HRS 50MINS	497	9

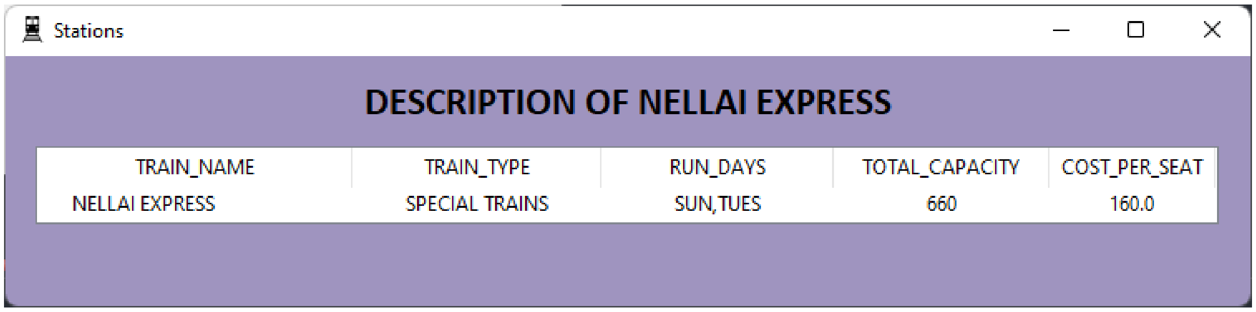
Fig 6. Station_details screen to display the stations a selected train passes through.


Stations

CHERAN EXPRESS PASSES THROUGH THE STATIONS

STATION_NAME	ARRIVAL	DEPARTURE	HALT_TIME	RUNNING_DAY
COIMBATORE JUNCTION	START	22-40	-	1
TIRUPPUR	23-18	23-20	2MINS	1
ERODE JUNCTION	00-05	00-10	5MINS	2
SALEM JUNCTION	00-57	01-00	3MINS	2
KATPADI JUNCTION	04-08	04-10	2MINS	2
ARAKKONAM JUNCTION	04-58	05-00	2MINS	2
PERAMBUR	05-49	05-50	1MIN	2

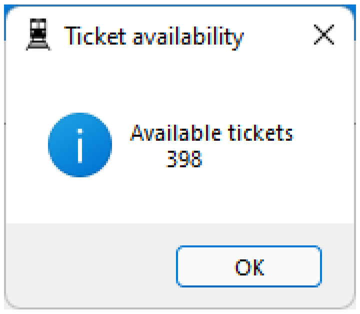
Fig 7. Train_desc screen to display the description of a selected train.


Stations

DESCRIPTION OF NELLAI EXPRESS

TRAIN_NAME	TRAIN_TYPE	RUN_DAYS	TOTAL_CAPACITY	COST_PER_SEAT
NELLAI EXPRESS	SPECIAL TRAINS	SUN,TUES	660	160.0

Fig 8. Ticket_availability screen to display the availability of tickets of a selected train.


Ticket availability

Available tickets
398

Appendix - Report on Python GUI tkinter.

Tkinter is the standard GUI library for Python. It is the Python interface to the Tk GUI toolkit shipped with Python.

To create a window in tkinter we user a Tk() object, which allows us to create a standalone window.

```
import Tkinter
top = Tkinter.Tk()
# Code to add widgets will go here...
top.mainloop()

The above code creates a blank window when executed.
```

Tkinter Widgets

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

Brief description of all the Tkinter widgets used in this project.

1. Label:

Label widget implements a display box where you can place text or images. The text displayed by this widget can be updated at any time.

Syntax:

```
l = Label ( master, option, ... )
```

Parameters:

- ***master*** – This represents the parent window.
- ***options*** – text, font, bg, image, justify, fg, bitmap etc.

2. Button:

The Button widget is used to add buttons in a Python application. These buttons can display text or images that convey the purpose of the buttons. You can attach a function or a method to a button which is called automatically when you click the button.

Syntax:

```
b = Button ( master, option=value, ... )
```

Parameters:

- ***master*** – This represents the parent window.
- ***options*** – text, font, bg, fg, image, height, width, command etc.

3. Frame:

The Frame widget is very important for the process of grouping and organizing other widgets in a somehow friendly way. It works like a container, which is responsible for arranging the position of other widgets.

Syntax:

```
l = Label ( master, option, ... )
```

Parameters:

- ***master*** – This represents the parent window.
- ***options*** – bg, height, cursor, bd, width, relief etc.

4. Canvas:

The Canvas is a rectangular area intended for drawing pictures or other complex layouts. Canvas can be used to place graphics, text, widgets or even frames.

Syntax:

```
c = Canvas ( master, option=value, ... )
```

Parameters:

- ***master*** – This represents the parent window.
- ***options*** – bd, bg, height, cursor, width, yscrollcommand, xscrollcommand etc.

5. Scrollbar:

This widget provides a slide controller that is used to implement vertical scrolled widgets, such as Listbox, Text and Canvas.

Syntax:

```
s = Scrollbar ( master, option, ... )
```

Parameters:

- ***master*** – This represents the parent window.
- ***options*** – bg, bd, command, orient, width, repeatinterval, takefocus etc.

6. Toplevel:

Toplevel widgets work as windows that are directly managed by the window manager. They do not necessarily have a parent widget on top of them.

Syntax:

```
t = Toplevel ( master, option, ... )
```

Parameters:

- ***master*** – This represents the parent window.
- ***options*** – bd, bg, cursor, height, width, relief, font, fg etc.

7. Treeview:

A treeview widget displays a hierarchy of items and allows users to browse through it. One or more attributes of each item can be displayed as columns to the right of the tree. It can be used to build user interfaces similar to the tree display you'd find in file managers.

Syntax:

```
tree = ttk.Treeview(master, option, ...)
```

Parameters:

- ***master*** – This represents the parent window.
- ***options*** – height, displaycolumns, padding, columns, selectmode etc.

Source: Tkinter python