Odd Semester (2020)



**BINUS UNIVERSITY**

**BINUS INTERNATIONAL**

**Assignment Cover Letter**

**(Individual Work****)**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | |  | |  | |
| **Student Information**: **Surname** | | | | | **Given Names**  **Lamin** | | **Student ID Number**  **2101711103** | |
| 1. | | **Gitteh** |  | |
|  |  |
| **Course Code** | **: COMP6502** |  |  | | **Course Name** | | **: Introduction to Programming** | |
| **Class** | **: L1AC** |  |  | | **Name of Lecturer(s)** | | **:** 1. Bagus Kerthyayana | |
|  |  |  |  | |  | | 2. | |
| **Major** | **: CS** |  |  | |  | |  | |
| **Title of Assignment**  (if any) | : Banking Simulation | |  |  | |  | |  | |
| **Type of Assignment**    **Submission Pattern** | **: Final Project** |  |  | |  | |  | |
| **Due Date** | **: 6-11-2016** |  |  | | **Submission Date** | | **: 6-11-2016** | |

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer’s instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

# Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

# Declaration of Originality

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: (Name of Student)

1. Lamin Lamin Gitteh

**“Banking Simulation”**

**Name : Lamin Gitteh**

**ID : 2101711103**

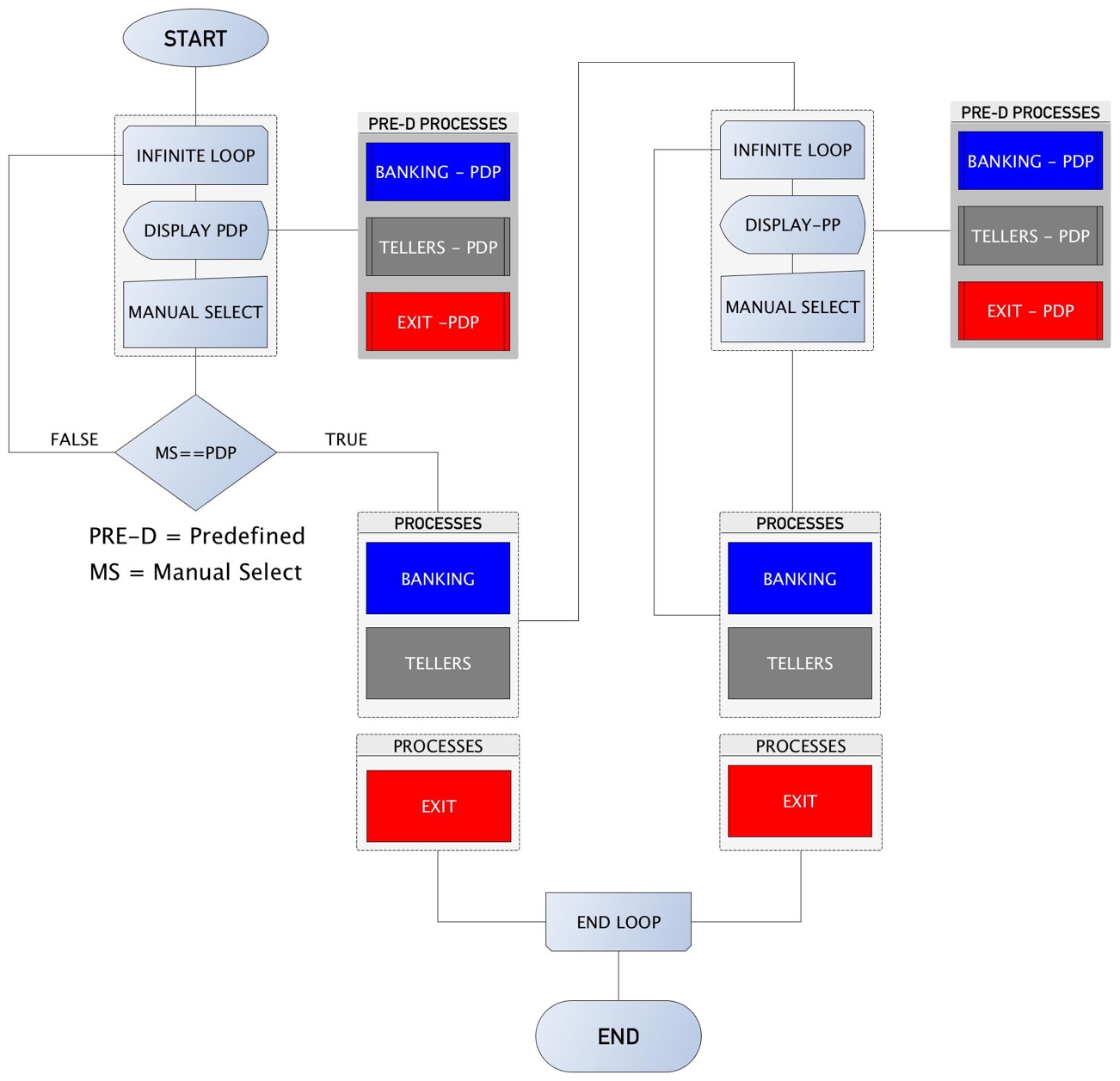
1. **Description**

**The function of this program:**

This project is a banking management simulation with deposit, add new account, remove, withdrawal, and transferring functionality. It consists of 4 python script files that are all connected to the same database helping the operator to manage any saved account.

1. Menu.py – Start
2. Operators – To manage operator/users
3. Simulation – The main program
4. Statistic – Matplotlib module

**II.a. Design/Plan**

**Project’s Flowchart**

**II.b. Explanation of Each function**

**Main.py: ( *def load()* )**

1. **Outside def load ():**

* Contains a time module which the clock variable uses to assign 1 second sleep to all the options available to the user before executing the command.
* A call statement to execute the def load() method.

1. **Inside def load ():**

* A list of variable named ‘services’ is defined and declared.
* Using a while to loop through the services list and show the options whether the user want to go to the ‘Run Banking’, ‘Manage Teller’ , or to ‘Exit the program’.
* If the user’s input is ‘Run Banking’, the program will import the ‘Simulation.py’ script and execute the script.
* If the user’s input is ‘Manage Teller’, the program will import the ‘Operators.py’ script and execute the script.
* Finally, If the user’s input is ‘Exit’, the program will be terminated.

**Operators.py: ( *class AllTellers()* )**

1. **Outside class AllTellers ():**

* Contains a time module which the clock variable uses to record time of certain task when executed.
* Sqllite3 module is declared and defined enabling the program to store and retrieve data within the database.
* Declare the database and cursor.
* A list holding names of all options a user can choose from.
* A declared function ‘def starter()’ with a while loop to print the list and show options to the user whether the user wants to choose from the available options.
* If the user chooses any of the options except the ‘Exit Application’, an instance of a the AllTellers class call run will be executed and calling any of the selected function in the AllTellers class.

1. **Inside class AllTellers ():**

* Contains a declared database statement – To create a new table if not exists and assign tellerid and password attributes.
* Def register() function to register new teller/user.
* Def remove() to remove any registered user from the database
* Def viewtellers() to display list of all registered users.
* Def change() to change user password.

**Simulation.py: (*class MyImports(),class Bank(),class Transactions()*)**

1. **Inside the classes ():**

* MyImports class contains functions with import statements.
* def menu() imports script file Menu.py
* def statistic() imports Statistic.py
* def tellers() imports the Operators.py script.
* Bank class inherits the MyImports class.
* The Bank class contains dictionary variable ‘services’ which holds names of options available for the users to operate from.
* An empty dictionary variable named ‘accountstransactions’ is defined to store account number and transaction.
* An empty list variable named ‘records’ to store transactions.
* A global login and logpass variable – which will be used to compare user’s login details if the user logged out.
* The Transactions class inherits Bank class to access all variables and methods in the Bank class.
* A global variable named ‘chosenservice’ is declared which is used as arguments for 'def choose’ function.
* The Transactions class constructor takes 2 arguments which are passed in from the instance of the Transactions class named ‘\_\_user’ passing in user’s valid login id and password before the program runs. The constructor calls self.start() function which is defined in the same class.
* def \_start() starts with a while loop accessing self.services variable which is in the Bank class and shows the list of options the user can choose from.
* The predefined global variable ‘chosenservice’ is declared in the def start() function asking for user integer input and then pass the value to self.choose() function as an argument.
* The def choose() function contains conditional if statements which executes based on the value chosenservice variable holds.
  + If chosenservice == 1, execute newaccount function.
  + If chosenservice == 2, execute deposit function
  + If chosenservice == 3, execute withdrawal function
  + If chosenservice == 4, execute transfer function
  + If chosenservice == 5, execute useraccount function
  + If chosenservice == 6, execute remove function
  + If chosenservice == 7, execute allrecords function
  + If chosenservice == 8, execute dailyincome function
  + If chosenservice == 9, execute incomestatistic
  + If chosenservice == 10, execute lockscreen function
  + If chosenservice == 11, exit the program.
* 1. The def **newaccount function** contains sqllite database connection execution statement.
  + connection = sqlite3.connect(“Bank.db”) connects the function to the database ‘Bank.db’
  + cursor = connection.cursor() carries execution statements within the database. It is through this statement the user can store or retrieve data from the database.
* 2. The def **deposit** **function** allows the user to increment customers account.
  + Surrounded with a try and except block to avoid crashing if user inputs unsupported character or error connecting to the database.
  + customeraccount = int(input()) takes integer value.
  + findaccount = 'SELECT \* FROM customers WHERE accnumber = ?' with a for loop, cursor.execute checks the findaccount statement against customeraccount. If available/true, breaks the loop then jump to customerdeposit = int(input()) takes integer value. If unavailable/false, the loop will jump to the else statement ‘No account found’.
  + getdepositaccount = cursor.fetchall() retrieves the column with customeraccount in list tuple.
  + deposittobalance = getdepositaccount [0][1] retrieves the 1st set and 2nd value in the set.
  + currentbalance = customerdeposit + deposittobalance will increment the customer’s balance by the value passed in the customerdeposit variable.
  + cursor.execute(“UPDATE customers SET accbalance = %d WHERE accnumber = %d % (currentbalance, customeraccount)) will update the database with customers balance with the currentbalance value.
  + connection.commit() – will save execution statement.
* 3. The **withdrawal** **function** allows the user to deduct from a customer’s account.
  + Surrounded with a try and except block to avoid crashing if user inputs unsupported character or error connecting to the database.
  + customeraccount = int(input()) takes integer value.
  + findaccount = 'SELECT \* FROM customers WHERE accnumber = ?' with a for loop, cursor.execute checks the findaccount statement against customeraccount. If available/true, breaks the loop then jump to customerwithdrawal = int(input()) takes integer value. If unavailable/false, the loop will jump to the else statement ‘No account found’.
  + getwithdrawalaccount = cursor.fetchall() fetch the column with customeraccount and return in list tuple.
  + customersbalance = getwithdrawalaccount [0][1] retrieves the 1st set and 2nd value in the set.
  + afterwithdrawn = customersbalance – customerwithdrawal
  + If condition checks the balance against the withdrawal value. If customerwithdrawal value is less than or equal to afterwithdrawn value, cursor.execute(“UPDATE customers SET accbalance = %d WHERE accnumber = %d % (afterwithdrawn, customeraccount)) will update the database with customers balance with the afterwithdrawn value. Else if afterwithdrawn is greater then custoberbalance, The condition jumps to “Your have insufficient balance”.
  + connection.commit() – will save execution statement.
* 4. The **transfer** **function** allows the user to transfer from a customer’s account.
  + Surrounded with a try and except block to avoid crashing if user inputs unsupported character or error connecting to the database.
  + transferfrom = int(input()) takes integer value.
  + findaccount = 'SELECT \* FROM customers WHERE accnumber = ?' with a for loop, cursor.execute checks the findaccount statement against transferfrom. If available/true, breaks the loop then jump to the next line cursor.execute(find,(transferfrom,)) which checks a row with transferfrom account number else execuse the for else statement which print out “No Account found”.
  + results = cursor.fetchall() fetch the row with transferfrom account\_number.
  + gettransferfromaccount = results [0][0] retrieves the 1st set and 1st value in the set which is the account number.
  + gettransferfrombalance = results [0][1] retrieves the 1st set and 2nd value in the set which is account balance.
  + Using a for loop and cursor.execute statement to check transferto = int(input()) account\_number in the database. If available, the loop breaks and jumps to

cursor.execute(findaccount,(transferto))

* + results = cursor.fetchall() fetch the row with transferto account\_number.
  + gettransfertoaccount = results [0][0] retrieves the 1st set and 1st value in the set which is the account number.
  + gettransfertobalance = results [0][1] retrieves the 1st set and 2nd value in the set which is account balance.
  + If condition checks the gettransferfrombalance against transferamount value. If transferamount value is less than or equal to gettransferfrombalance value, transfertocurrentbalance = gettransferfrombalance – transferamount, transfertocurrentbalance = gettransfertobalance + transferamount and cursor.execute(“UPDATE customers SET accbalance = %d WHERE accnumber = %d % (transfromcurrentbalance, transferfrom)) and (“UPDATE customers SET accbalance = %d WHERE accnumber = %d % (transfertocurrentbalance, transferto)) this will update both account\_balance with the new balance.
  + connection.commit() – will save execution statement.
  + If transferto not available, the loop breaks and jumps to transferamont which ask for user integer(input).
  + charges = ((transferamoutn \*5)/100) holds a 5% of transferamount to be use as charges for account\_number not in the bank database.
  + totalcharges = transferamount + charges – which adds up the transferamount and the charges value.
  + If condition checks the gettransferfrombalance against totalcharges value. If totalcharges value is less than or equal to gettransferfrombalance value, transfertocurrentbalance = gettransferfrombalance – transferamount and cursor.execute(“UPDATE customers SET accbalance = %d WHERE accnumber = %d % (transfromcurrentbalance, transferfrom))
  + connection.commit() – will save execution statement.
  + Else if totalcharges is greater then gettransferfrombalance, The condition jumps to “Your have insufficient balance”.
* 5. The **useraccount** **function** allows the user to check customer’s account.
  + Surrounded with a try and except block to avoid crashing if user inputs unsupported character or error connecting to the database.
  + uanumber = int(input()) takes integer value which is expected to be customer’s account number.
  + check = 'SELECT \* FROM customers WHERE accnumber = ?' with a for loop, cursor.execute checks the check statement against uanumber. If available/true, breaks the loop then jump to the cursor.execute(find,( uanumber,)) which checks a row with uanumber account number else execuse the for else statement which print out “No Account found”.
  + balanceresult = cursor.fetchall() fetch the row with uanumber account\_number.
  + uaresult = balanceresult[0][0] retrieves the 1st set and 1st value in the set which is the account number.
  + Print uaresult ‘The customer’s balance” is executed.
* 6. The **remove** **function** allows the user to check customer’s account.
  + Surrounded with a try and except block to avoid crashing if user inputs unsupported character or error connecting to the database.
  + ranumber = int(input()) takes integer value which is expected to be customer’s account number.
  + check = 'SELECT \* FROM customers WHERE accnumber = ?' with a for loop, cursor.execute checks the check statement against renumber. If available/true, breaks the loop then jump to the cursor.execute(“DELETE FROM customers WHERE accnumber = ?”, (ranumber,)) which checks a row with ranumber account number and delete the row from the database.
  + connection.commit() – will save execution statement.
  + Else if ranumber not found, the for loop executes the else statement which print out “No Account found”.
* 7. The **allrecords** **function** prints all the logs.
  + Surrounded with a try and except block to avoid crashing if user inputs unsupported character.
  + Using a for loop to print values in self.record list which appends information when a new account is added, deposits, withdrawn, transferred or removed.
* 8. The **dailyincome** **function** prints the total income.
  + The function begins with if conditions statement : if self.income != 0 (not equal to 0) then show the self.income value and it’s submenu else print out “Your current income is 0”.
  + The submenu is surrounded with try a try block statement to avoid crashing when user inputs invalid characters. Using a for loop to print values in list variable named incomelist which holds available options for user to choose from.
  + 1 = Maximum Transaction – Returns the maximum value in dictionary self.accountstransaction which holds all the value user processed.

maximum = max (self.accountstransactions, key =self.accountstransactions.get)

max (dictionary variable, key = get key with the highest value)

* + 2 = Minimum Transaction – Returns the minimum value in dictionary self.accountstransaction which holds all the value user processed.

max (dictionary variable, key = get key with the highest value)

* + 3 = calls the def start() functions.
* 9. The **incomestatistic function** retrieves all income data saved in the salesrecord table within the database.
  + connection = sqlite3.connect(“Bank.db”) connects the function to the database ‘Bank.db’
  + cursor = connection.cursor() carries execution statements within the database. It is through this statement the user can store or retrieve data from the database.
  + Using a for loop to print values in private list variable named submenu which will present to the user options to choose from.
  + If user input == 1 (“Update Statistic & Run”), cursor.execute(“SELECT count(\*) FROM salesrecords”) will set the cursor to get get the row count of salesrecords.
* self.showlastrow = cursor.fetachall() gets all the column in the table and assign all to self.showlastrow.
* self.lastrow = self.showlastrow [0][0] + gets the number of row in the table and 1st set,1st value in the set and increment it by 1.
* self.counter = “day”+str(self.lastrow) converts the self.lastrow to string variable and concatenate it with “day” to auto increment the days value in the salesrecords table of the database.
* cursor.execute("INSERT INTO salesrecords VALUES ('%s','%d')" % (self.counter, self.income)) inserts new data into the salesrecords table with the self.conter and self.income values.
* connection.commit() function saves the execution/statement.
* Finally, self.satistic() function will be called with the updated income value as the last day.
* If user input == 2 (“View Current Statistic”), self.statistic will be called without the current income.
* If user input == 3 (“Back to Menu”), def start() function will be called which will loop through the services for user to choose from.
* 10. The **lockscreen function** locks the session for the user.
* Starts with a while loop asking user for login id and password input.
* If user’s input id and password equals to global login and global logpass, instance of the Transaction class will be called passing login and password values as the argument.
* If user’s input id and password different from the global login details, the program will give the user “No user or wrong password message”
* 11. The **exitapplication function** exits the application.
  + User is prompt with a confirmation message to either save the income before exiting the application or not.
  + If user inputs is equal yes, connection = sqlite3.connect(“Bank.db”) is declared which connects the function to the database ‘Bank.db’
  + cursor = connection.cursor() is defined which carries execution statements within the database. cursor.execute(“SELECT count(\*) FROM salesrecords”) will set the cursor to get the row count of salesrecords.
* self.showlastrow = cursor.fetachall() gets all the column in the table and assign all to self.showlastrow.
* self.lastrow = self.showlastrow [0][0] + gets the number of row in the table and 1st set,1st value in the set and increment it by 1.
* self.counter = “day”+str(self.lastrow) converts the self.lastrow to string variable and concatenate it with “day” to auto increment the days value in the salesrecords table of the database.
* cursor.execute("INSERT INTO salesrecords VALUES ('%s','%d')" % (self.counter, self.income)) inserts new data into the salesrecords table with the self.conter and self.income values.
* connection.commit() function saves the execution/statement.

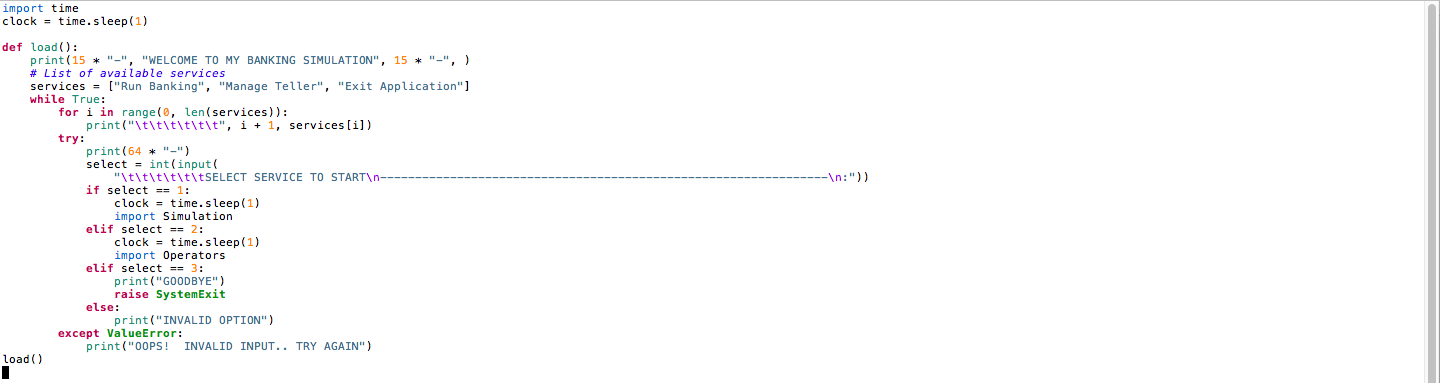
1. **Outside MyImports, Bank and Transactions class ():**

* Global login and global logpass is declared to be accessed by the lock function.
* Using a while true loop and try except block to ask ask for user input if user input is incorrect.
* login = input(“Enter ID”) ask for user login id
* logpass = input(“Enter password”) ask for user password.
* find\_user = 'SELECT \* FROM operators WHERE tellerid = ? AND password = ?'
* cursor.execute checks the find\_user statement against login and logpass in the database.
* results = cursor.fetchall() fetch row with login and logpass value from the operators table within the “Bank.db” database.
* If statement to verify the login details and if ‘results’ which is the same as if available in the table, then execute statement.

**Statistic.py:**

* Contains a matplotlib module which displays the chart
* Sqlite3 module to link the program with a database
* connection = Sqlite3.connect(“Bank”) connects the program with the “Bank.db” database.
* cursor = connection.cursor() is declared to store and retrieve for days column.
* cursorforrow. connection.cursor() is declared to store and retrieve for data column in.
* days = [] is an empty list to hold day values when the for loop appends data.
* days = [] is an empty list to hold dayincome values when the for loop appends data.
* getrow = "SELECT count(\*) FROM salesrecords" = Is a database execution statement to get all row count.
* getdata = "SELECT dayincome FROM salesrecords" = Is a database execution statement to return all values in dayincome columns within salesrecords table.
* cursor.execute(getdata) execute the statement to get all values in dayincome column within salesrecords table.
* cursorforrow.execute(getrow) execute the statement to get the length of rows in the salesrecords table.
* incomedata = cursor.fetchall() retrieves dayincome values.
* rowcount = cursorforrow.fetchall() retrieves length of the row.
* num = int(rowcount[0][0]) is an integer variable to get the 1st value in the 1st set of rowcount variable.
* for i in range(0, num) loops from 0 to the len of variable num value executing days.append("Day "+str(i+1)) which formats variable ‘i’ from int to string then concatenate with days to auto increment and append the value to days list.
* data.append(incomedata[i]) appends the value in the incomedata column following indexing based on value of variable ‘i’.
* day = [x for x in range(len(days))] loops through the days list and appends the index value to day.
* plt.plot(data, linewidth = 5) = draws graphs base on the content of the data list with a linewidth of 5px.
* plt.title("S-B STATISTICS", fontsize = 24) ) represents the title of the graph/program.
* plt.show() displays the graph/program.

**Menu.py**

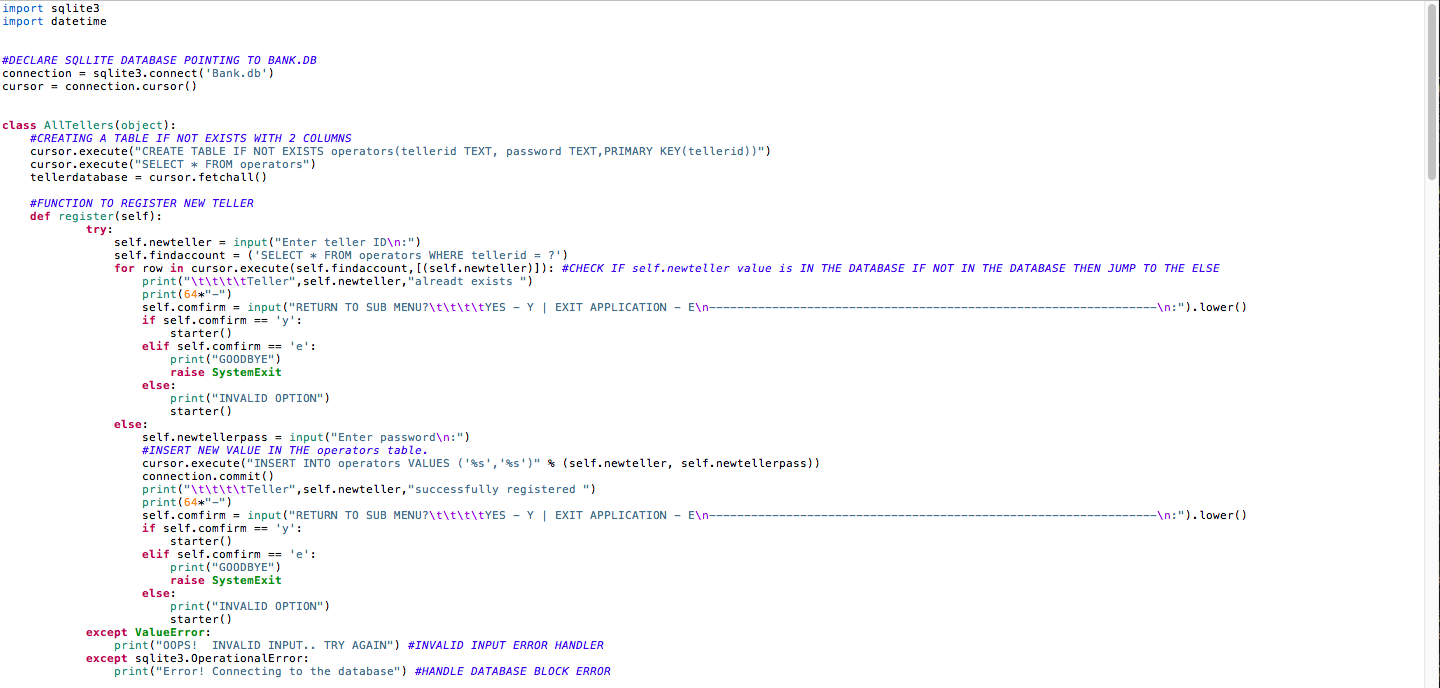
****

**Operators.py**

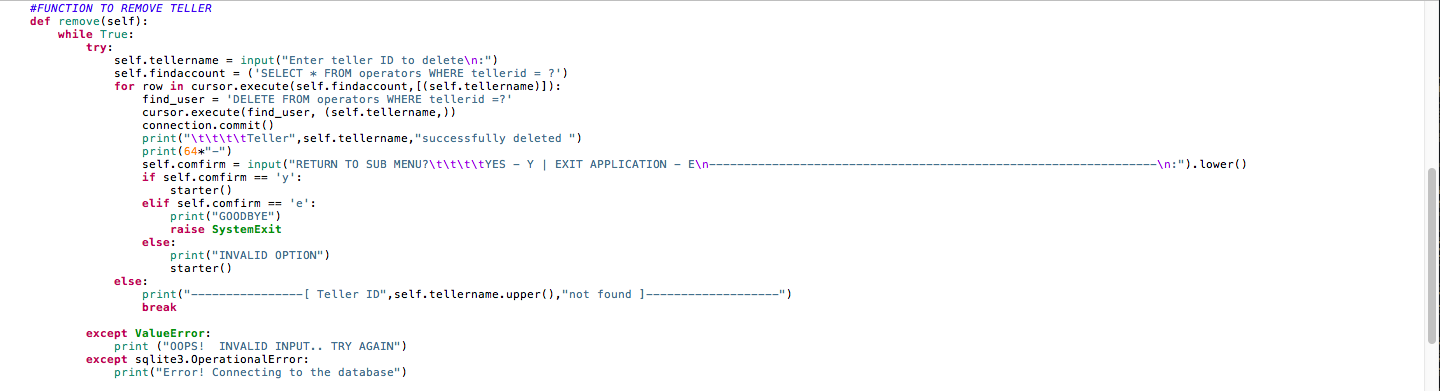
**def starter()**

****

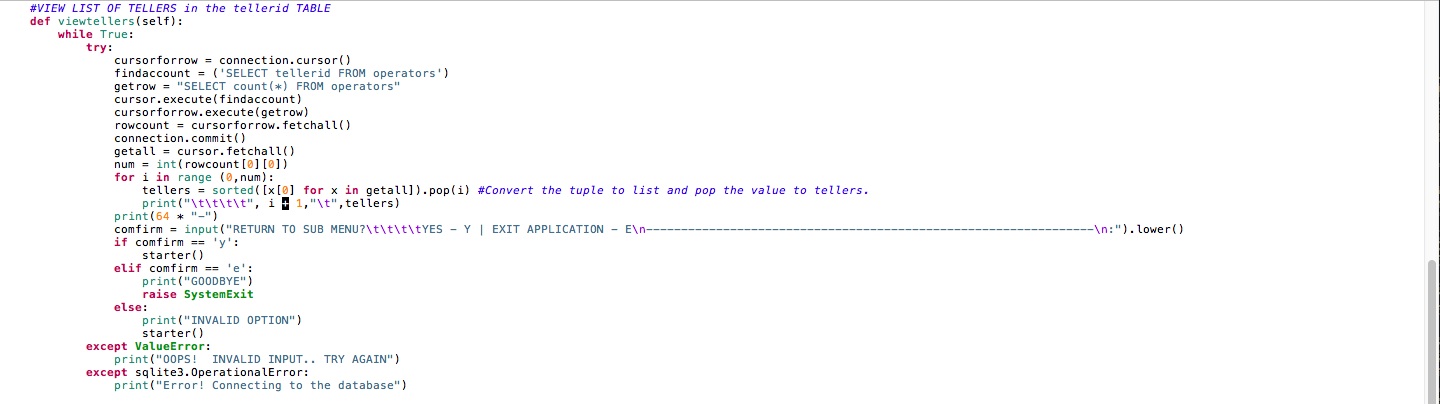
**Operators register()**

****

**Operators remove()**

****

**Operators viewtellers()**

****

**Operators change()**

****

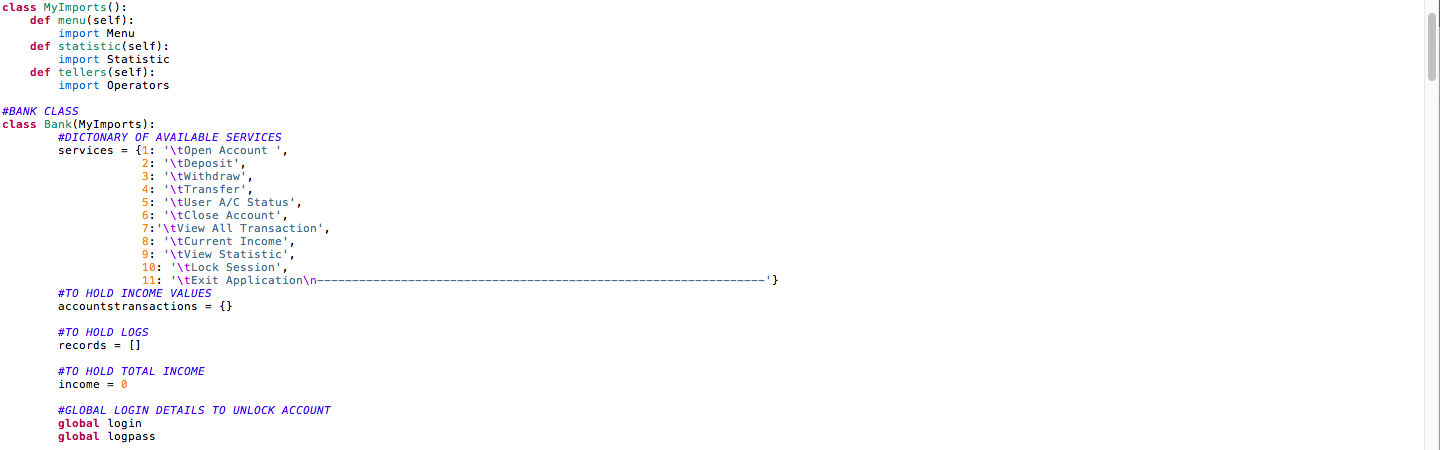
**Simulation.py**

**Modules**

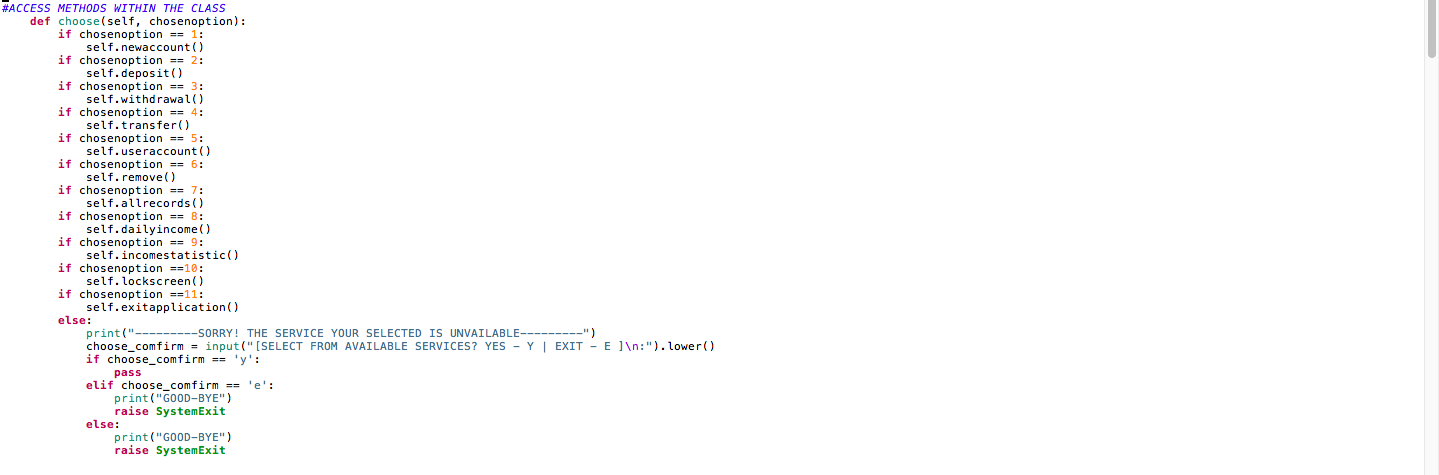
****

**Start…………**

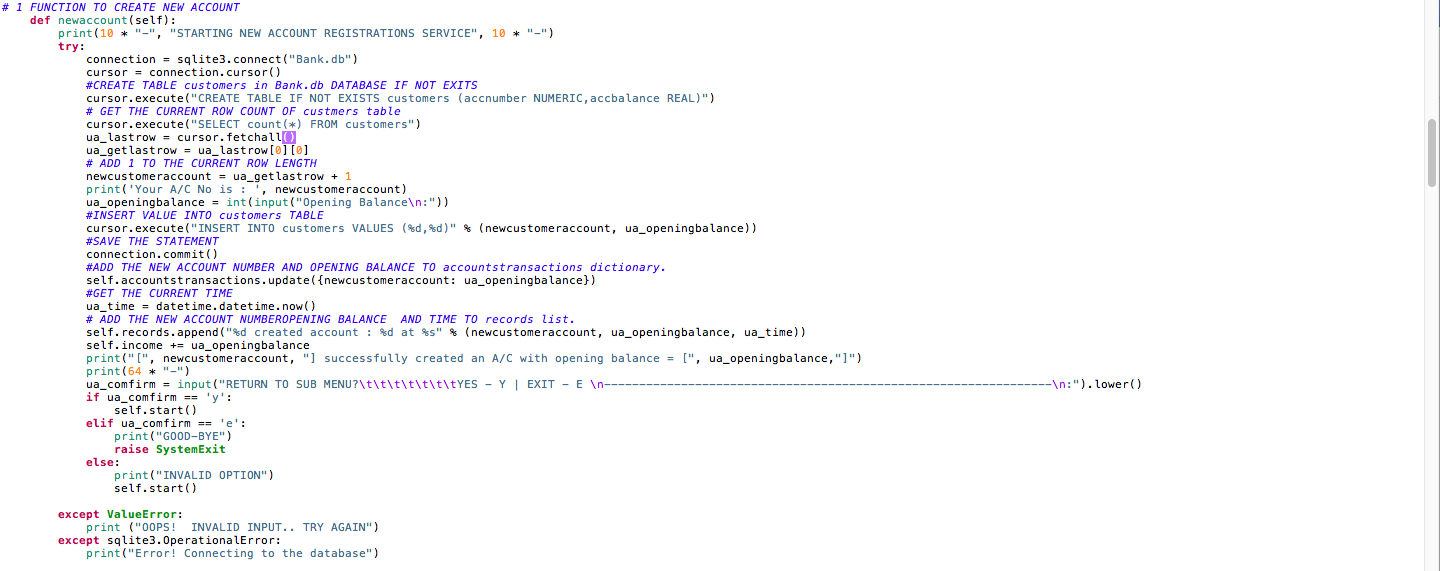
****

**Imports & Bank class()**

**Transactions class**

**Transactions choose()**

**Transactions newaccount()**

****

**Transactions deposit()**

****

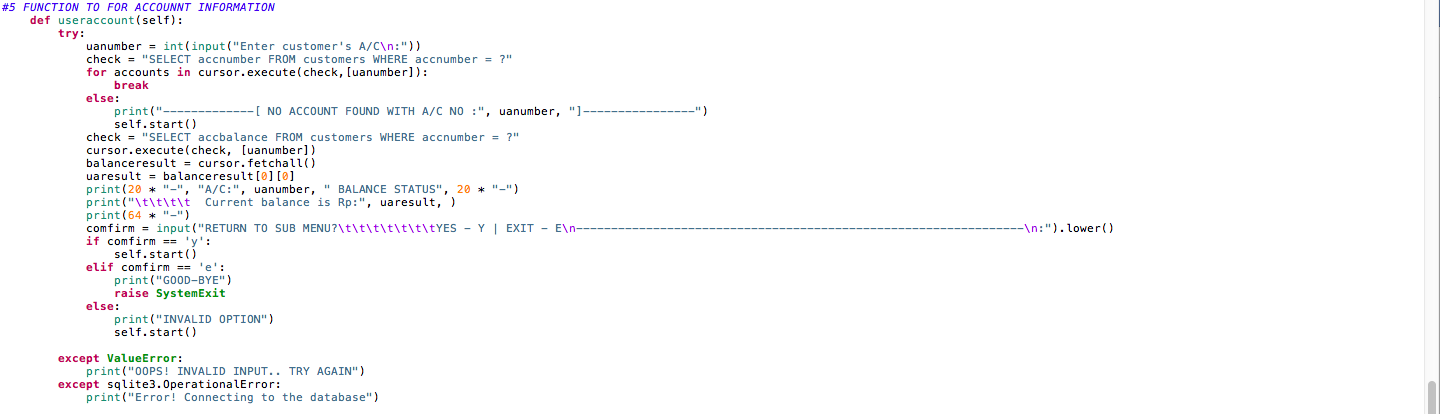
**Transactions withdrawal()**

****

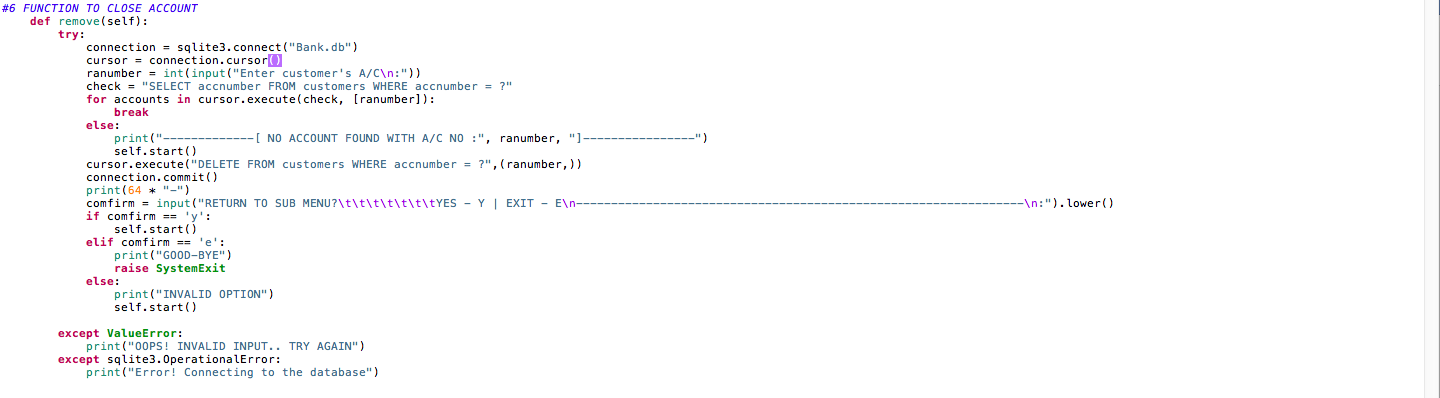
**Transactions transfer()**

****

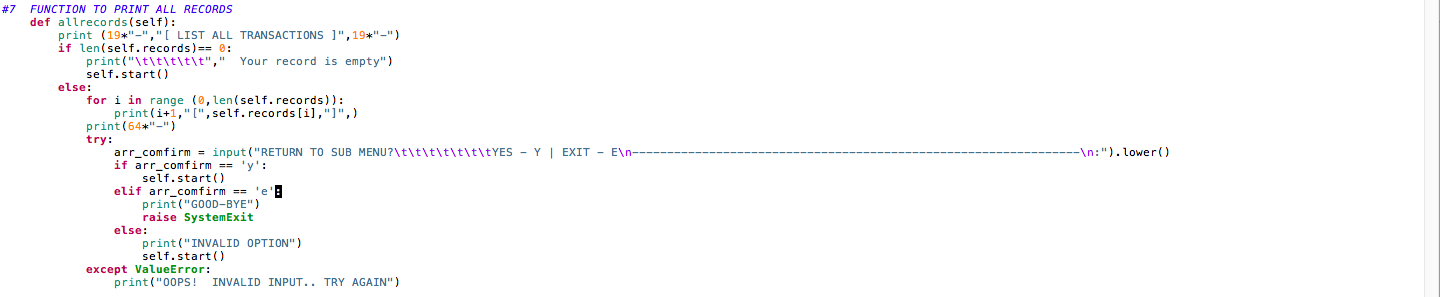
**Transactions useraccount()**

****

**Transactions remove()**

****

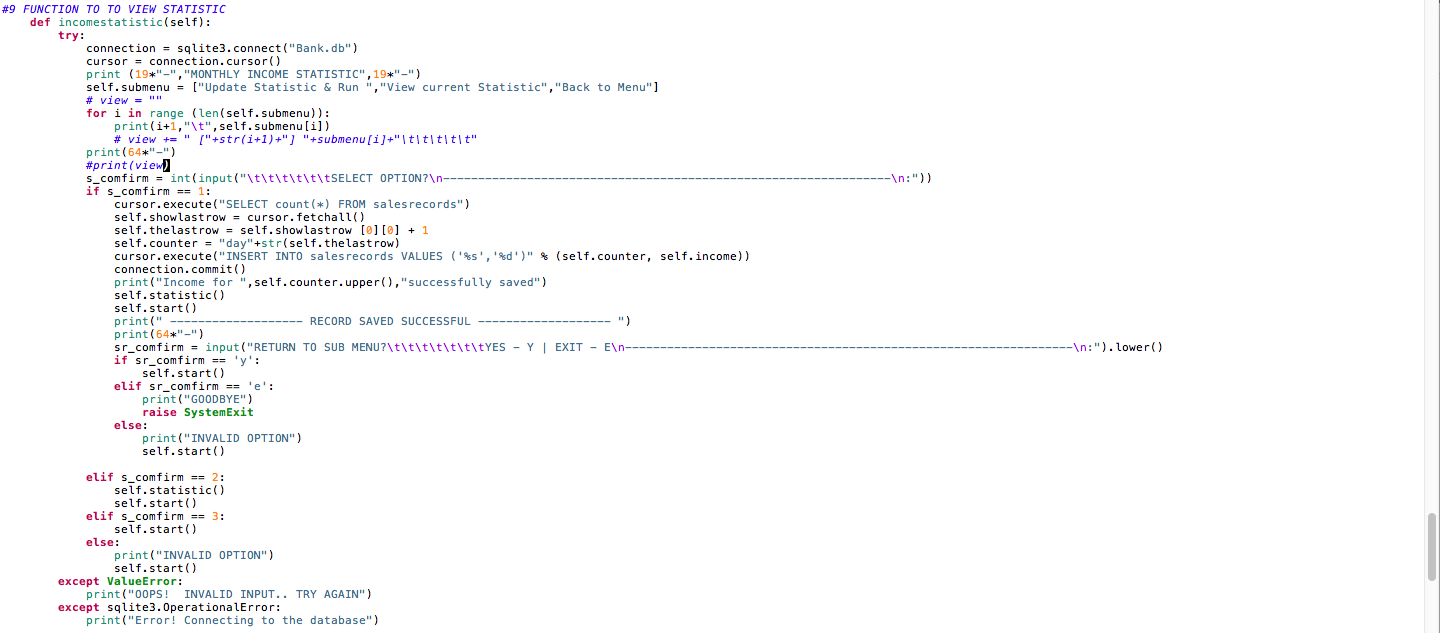
**Transactions allrecords()**

****

**Transactions dailyincome()**

****

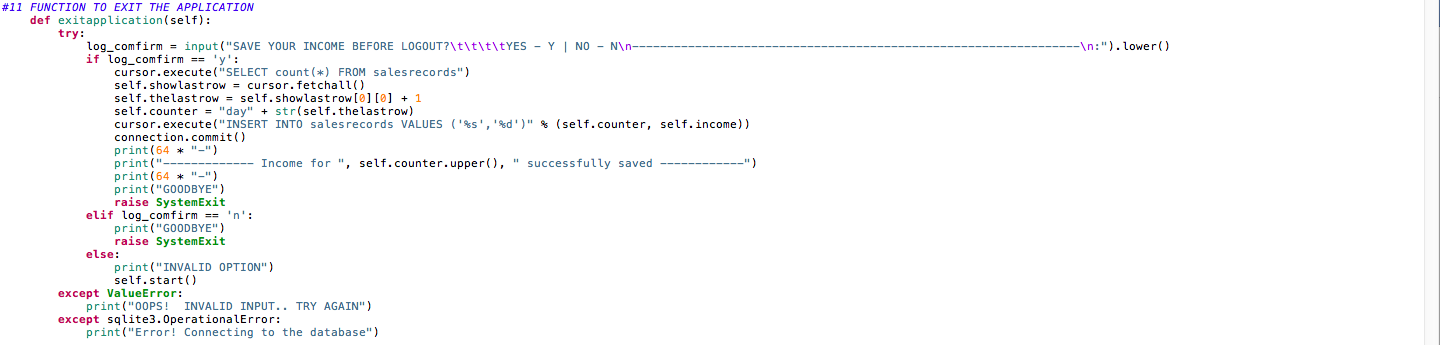
**Transactions incomestatistic ()**

****

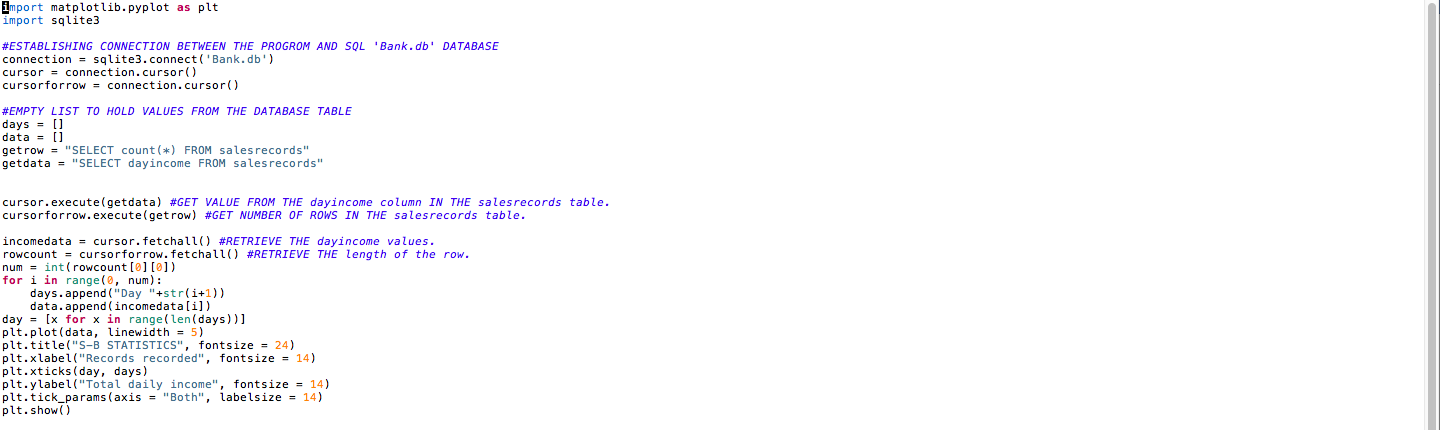
**Transactions lock ()**

****

**Transactions exitapplication ()**

****

**Statistics.py**

****

**SOURCE CODES**

**Menu.py Source Code**

**…………………………………………………………………………………………………….**

import time

clock = time.sleep(1)

def load():

print(15 \* "-", "WELCOME TO MY BANKING SIMULATION", 15 \* "-", )

# List of available services

services = ["Run Banking", "Manage Teller", "Exit Application"]

while True:

for i in range(0, len(services)):

print("\t\t\t\t\t\t", i + 1, services[i])

try:

print(64 \* "-")

select = int(input(

"\t\t\t\t\t\tSELECT SERVICE TO START\n----------------------------------------------------------------\n:"))

if select == 1:

clock = time.sleep(1)

import Simulation

elif select == 2:

clock = time.sleep(1)

import Operators

elif select == 3:

print("GOODBYE")

raise SystemExit

else:

print("INVALID OPTION")

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN")

load()

**…………………………………………………………………………………………………….**

**Operators.py Source Code**

**…………………………………………………………………………………………………….**

import sqlite3

import datetime

#DECLARE SQLLITE DATABASE POINTING TO BANK.DB

connection = sqlite3.connect('Bank.db')

cursor = connection.cursor()

class AllTellers(object):

#CREATING A TABLE IF NOT EXISTS WITH 2 COLUMNS

cursor.execute("CREATE TABLE IF NOT EXISTS operators(tellerid TEXT, password TEXT,PRIMARY KEY(tellerid))")

cursor.execute("SELECT \* FROM operators")

tellerdatabase = cursor.fetchall()

#FUNCTION TO REGISTER NEW TELLER

def register(self):

try:

self.newteller = input("Enter teller ID\n:")

self.findaccount = ('SELECT \* FROM operators WHERE tellerid = ?')

for row in cursor.execute(self.findaccount,[(self.newteller)]): #CHECK IF self.newteller value is IN THE DATABASE IF NOT IN THE DATABASE THEN JUMP TO THE ELSE

print("\t\t\t\tTeller",self.newteller,"alreadt exists ")

print(64\*"-")

self.comfirm = input("RETURN TO SUB MENU?\t\t\t\tYES - Y | EXIT APPLICATION - E\n-----------------------\n:").lower()

if self.comfirm == 'y':

starter()

elif self.comfirm == 'e':

print("GOODBYE")

raise SystemExit

else:

print("INVALID OPTION")

starter()

else:

self.newtellerpass = input("Enter password\n:")

#INSERT NEW VALUE IN THE operators table.

cursor.execute("INSERT INTO operators VALUES ('%s','%s')" % (self.newteller, self.newtellerpass))

connection.commit()

print("\t\t\t\tTeller",self.newteller,"successfully registered ")

print(64\*"-")

self.comfirm = input("RETURN TO SUB MENU?\t\t\t\tYES - Y | EXIT APPLICATION - E\n------------------\n:").lower()

if self.comfirm == 'y':

starter()

elif self.comfirm == 'e':

print("GOODBYE")

raise SystemExit

else:

print("INVALID OPTION")

starter()

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN") #INVALID INPUT ERROR HANDLER

except sqlite3.OperationalError:

print("Error! Connecting to the database") #HANDLE DATABASE BLOCK ERROR

#FUNCTION TO REMOVE TELLER

def remove(self):

while True:

try:

self.tellername = input("Enter teller ID to delete\n:")

self.findaccount = ('SELECT \* FROM operators WHERE tellerid = ?')

for row in cursor.execute(self.findaccount,[(self.tellername)]):

find\_user = 'DELETE FROM operators WHERE tellerid =?'

cursor.execute(find\_user, (self.tellername,))

connection.commit()

print("\t\t\t\tTeller",self.tellername,"successfully deleted ")

print(64\*"-")

self.comfirm = input("RETURN TO SUB MENU?\t\t\t\tYES - Y | EXIT APPLICATION - E\n---------------\n:").lower()

if self.comfirm == 'y':

starter()

elif self.comfirm == 'e':

print("GOODBYE")

raise SystemExit

else:

print("INVALID OPTION")

starter()

else:

print("----------------[ Teller ID",self.tellername.upper(),"not found ]-------------------")

break

except ValueError:

print ("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

#VIEW LIST OF TELLERS in the tellerid TABLE

def viewtellers(self):

while True:

try:

cursorforrow = connection.cursor()

findaccount = ('SELECT tellerid FROM operators')

getrow = "SELECT count(\*) FROM operators"

cursor.execute(findaccount)

cursorforrow.execute(getrow)

rowcount = cursorforrow.fetchall()

connection.commit()

getall = cursor.fetchall()

num = int(rowcount[0][0])

for i in range (0,num):

tellers = sorted([x[0] for x in getall]).pop(i) #Convert the tuple to list and pop the value to tellers.

print("\t\t\t\t", i + 1,"\t",tellers)

print(64 \* "-")

comfirm = input("RETURN TO SUB MENU?\t\t\t\tYES - Y | EXIT APPLICATION - E\n---------------------------\n:").lower()

if comfirm == 'y':

starter()

elif comfirm == 'e':

print("GOODBYE")

raise SystemExit

else:

print("INVALID OPTION")

starter()

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

#FUNCTION TO CHANGE TELLER PASS

def change(self):

while True:

try:

print(14\*"-"," ENTER YOUR TELLER ID AND PASSWORD",14\*"-")

telleraccount = input("Enter teller ID:\n")

findaccount = ('SELECT \* FROM operators WHERE tellerid = ?')

for tellercheck in cursor.execute(findaccount,[(telleraccount)]):

break

else:

print("-------------[ NO ACCOUNT FOUND FOR :",telleraccount.upper(),"]----------------")

starter()

findpassword = ('SELECT \* FROM operators WHERE password = ?')

tellerspass = str(input("Enter teller password:\n"))

cursor.execute(findpassword, [(tellerspass)])

foundteller = cursor.fetchall()

theteller = foundteller[0][1]

if tellerspass == theteller: # Comparing password

pass1 = str(input("Enter new password:\n"))

pass2 = str(input("Confirm new password:\n"))

if pass1 == pass2:

updating = '''UPDATE operators SET password = ? WHERE tellerid = ?'''

cursor.execute(updating,(pass1,telleraccount))

connection.commit()

print("\t\t\t\tCongrats", telleraccount, " password successfully changed")

print(64 \* "-")

cp\_comfirm = input(

"RETURN TO SUB MENU?\t\t\t\tYES - Y | EXIT APPLICATION - E\n------------------------------------------\n:").lower()

if cp\_comfirm == 'y':

starter()

elif cp\_comfirm == 'e':

print("GOODBYE")

raise SystemExit

else:

print("INVALID OPTION")

starter()

else:

print("PASSWORD DOES NOT MATCH - TRY AGAIN")

break

except ValueError:

print ("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

print(8\*"-"," WELCOME TO MY BANKING REGISTRATION SIMULATION",8\*"-",)

print(64\*"-")

menu = ["Add new teller","View tellers","Change Teller Password","Remove teller","Main Menu","Exit Application"]

def starter():

while True:

try:

for i in range(0,len(menu)):

print(i+1,"",menu[i])

print(64\*"-")

select = int(input("\t\t\t\t\t\tSELECT SERVICE TO START\n----------------------------------------------------------------\n:"))

if select == 1:

run = AllTellers().register()

elif select == 2:

run = AllTellers().viewtellers()

elif select == 3:

run = AllTellers().change()

elif select == 4:

run = AllTellers().remove()

elif select == 5:

import Menu

elif select == 6:

print("GOODBYE")

raise SystemExit

else:

print("CHOOSE A VALID OPTION")

except ValueError:

print ("OOPS! INVALID INPUT.. TRY AGAIN")

starter()

**…………………………………………………………………………………………………….**

**Simulation.py Source Code**

**…………………………………………………………………………………………………….**

import datetime

import sqlite3

import os

chosenservice = 0

connection = sqlite3.connect('Bank.db')

cursor = connection.cursor()

class MyImports():

def menu(self):

import Menu

def statistic(self):

import Statistic

def tellers(self):

import Operators

#BANK CLASS

class Bank(MyImports):

#DICTONARY OF AVAILABLE SERVICES

services = {1: '\tOpen Account ',

2: '\tDeposit',

3: '\tWithdraw',

4: '\tTransfer',

5: '\tUser A/C Status',

6: '\tClose Account',

7:'\tView All Transaction',

8: '\tCurrent Income',

9: '\tView Statistic',

10: '\tLock Session',

11: '\tExit Application\n----------------------------------------------------------------'}

#TO HOLD INCOME VALUES

accountstransactions = {}

#TO HOLD LOGS

records = []

#TO HOLD TOTAL INCOME

income = 0

#GLOBAL LOGIN DETAILS TO UNLOCK ACCOUNT

global login

global logpass

#TRANSACTIONS CLASS INHERITING BANK CLASS

class Transactions(Bank):

global chosenservice

def \_\_init\_\_(self,operatorid,password):

self.start()

def start(self):

print(64\*"-")

while True:

for i in self.services:

print(i,self.services[i])

try:

chosenservice = int(input("\t\t\t\t\tCHOOSE SERVICE TO START\n----------------------------------------------------------------\n:"))

self.choose(chosenservice)

except ValueError:

print ("OOPS! INVALID INPUT.. TRY AGAIN")

#ACCESS METHODS WITHIN THE CLASS

def choose(self, chosenoption):

if chosenoption == 1:

self.newaccount()

if chosenoption == 2:

self.deposit()

if chosenoption == 3:

self.withdrawal()

if chosenoption == 4:

self.transfer()

if chosenoption == 5:

self.useraccount()

if chosenoption == 6:

self.remove()

if chosenoption == 7:

self.allrecords()

if chosenoption == 8:

self.dailyincome()

if chosenoption == 9:

self.incomestatistic()

if chosenoption ==10:

self.lockscreen()

if chosenoption ==11:

self.exitapplication()

else:

print("---------SORRY! THE SERVICE YOUR SELECTED IS UNVAILABLE---------")

choose\_comfirm = input("[SELECT FROM AVAILABLE SERVICES? YES - Y | EXIT - E ]\n:").lower()

if choose\_comfirm == 'y':

pass

elif choose\_comfirm == 'e':

print("GOOD-BYE")

raise SystemExit

else:

print("GOOD-BYE")

raise SystemExit

# 1 FUNCTION TO CREATE NEW ACCOUNT

def newaccount(self):

print(10 \* "-", "STARTING NEW ACCOUNT REGISTRATIONS SERVICE", 10 \* "-")

try:

connection = sqlite3.connect("Bank.db")

cursor = connection.cursor()

#CREATE TABLE customers in Bank.db DATABASE IF NOT EXITS

cursor.execute("CREATE TABLE IF NOT EXISTS customers (accnumber NUMERIC,accbalance REAL)")

# GET THE CURRENT ROW COUNT OF custmers table

cursor.execute("SELECT count(\*) FROM customers")

ua\_lastrow = cursor.fetchall()

ua\_getlastrow = ua\_lastrow[0][0]

# ADD 1 TO THE CURRENT ROW LENGTH

newcustomeraccount = ua\_getlastrow + 1

print('Your A/C No is : ', newcustomeraccount)

ua\_openingbalance = int(input("Opening Balance\n:"))

#INSERT VALUE INTO customers TABLE

cursor.execute("INSERT INTO customers VALUES (%d,%d)" % (newcustomeraccount, ua\_openingbalance))

#SAVE THE STATEMENT

connection.commit()

#ADD THE NEW ACCOUNT NUMBER AND OPENING BALANCE TO accountstransactions dictionary.

self.accountstransactions.update({newcustomeraccount: ua\_openingbalance})

#GET THE CURRENT TIME

ua\_time = datetime.datetime.now()

# ADD THE NEW ACCOUNT NUMBEROPENING BALANCE AND TIME TO records list.

self.records.append("%d created account : %d at %s" % (newcustomeraccount, ua\_openingbalance, ua\_time))

self.income += ua\_openingbalance

print("[", newcustomeraccount, "] successfully created an A/C with opening balance = [", ua\_openingbalance,"]")

print(64 \* "-")

ua\_comfirm = input("RETURN TO SUB MENU?\t\t\t\t\t\t\tYES - Y | EXIT - E \n-------------------------------------\n:").lower()

if ua\_comfirm == 'y':

self.start()

elif ua\_comfirm == 'e':

print("GOOD-BYE")

raise SystemExit

else:

print("INVALID OPTION")

self.start()

except ValueError:

print ("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

# 2 FUNCTION TO CREATE TO DEPOSIT

def deposit(self):

while True:

try:

connection = sqlite3.connect("Bank.db")

cursor = connection.cursor()

print(13 \* "-", "STARTING CUSTOMER DEPOSITING SERVICE", 13 \* "-")

customeraccount = int(input("Enter account number to deposit:\n"))

findaccount = 'SELECT \* FROM customers WHERE accnumber = ?'

for row in cursor.execute(findaccount,(customeraccount,)):

break

else:

print("-------------[ NO ACCOUNT FOUND WITH A/C NO :", customeraccount, "]----------------")

self.start()

findaccount = ('SELECT \* FROM customers WHERE accnumber = ?')

cursor.execute(findaccount, [(customeraccount)])

customerdeposit = int(input("Enter deposit amount:\n"))

getdepositaccount = cursor.fetchall()

deposittobalance = getdepositaccount[0][1]

currentbalance = customerdeposit + deposittobalance

cursor.execute("UPDATE customers SET accbalance = %d WHERE accnumber = %d" % (currentbalance, customeraccount))

connection.commit()

time = datetime.datetime.now()

print("--------[ A/C No :", customeraccount, "successfully deposited : Rp",customerdeposit, "]---------")

self.records.append("%d deposited : %d at %s" % (customeraccount, customerdeposit, time))

self.accountstransactions.update({customeraccount: customerdeposit})

self.income += customerdeposit

print(64 \* "-")

comfirm = input("RETURN TO SUB MENU?\t\t\t\t\t\t\tYES - Y | EXIT - E\n-----------------------------------\n:").lower()

if comfirm == 'y':

self.start()

elif comfirm == 'e':

print("GOOD-EYE")

raise SystemExit

else:

print("INVALID OPTION")

self.start()

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

#3 FUNCTION TO WITHDRAW

def withdrawal(self):

while True:

try:

connection = sqlite3.connect("Bank.db")

cursor = connection.cursor()

print(13 \* "-", "STARTING CUSTOMER WITHDRAWAL SERVICE", 13 \* "-")

customeraccount = int(input("Enter account number to withdraw from:\n"))

findaccount = ('SELECT \* FROM customers WHERE accnumber = ?')

for row in cursor.execute(findaccount, (customeraccount,)):

break

else:

print("-------------[ NO ACCOUNT FOUND WITH A/C NO :", customeraccount, "]----------------")

self.start()

cursor.execute(findaccount, (customeraccount,))

customerwithdrawal = int(input("Enter withdraw amount:\n"))

getwithdrawalaccount = cursor.fetchall()

customersbalance = getwithdrawalaccount[0][1]

afterwithdrawn = customersbalance - customerwithdrawal

if customersbalance >= afterwithdrawn:

cursor.execute("UPDATE customers SET accbalance = %d WHERE accnumber = %d" % (afterwithdrawn,customeraccount))

connection.commit()

time = datetime.datetime.now()

print("-------[ A/C No :", customeraccount,"successfully withdrawn : Rp",customerwithdrawal,"]---------")

self.records.append("%d withdrawn : %d at %s" % (customeraccount, customerwithdrawal, time))

self.income += customerwithdrawal

print(64 \* "-")

comfirm = input("RETURN TO SUB MENU?\t\t\t\t\t\t\tYES - Y | EXIT - E \n-------------------------------------------\n:").lower()

if comfirm == 'y':

self.start()

elif comfirm == 'n':

print("GOOD-BYE")

raise SystemExit

else:

print("INVALID OPTION")

self.start()

elif customersbalance <= afterwithdrawn:

print("YOUR HAVE INSUFFICIENT BALANCE")

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

# 4 FUNCTION TO TRANSFER

def transfer(self):

print(12 \* "-", "STARTING CUSTOMER TRANSFERRING SERVICE", 12 \* "-")

while True:

connection = sqlite3.connect("Bank.db")

cursor = connection.cursor()

try:

transferfrom = int(input("Enter account number to transfer from:\n"))

findaccount = 'SELECT \* FROM customers WHERE accnumber = ?'

for row in cursor.execute(findaccount, (transferfrom,)):

break

else:

print("-------------[ NO ACCOUNT FOUND FOR A/C NO :", transferfrom, "]----------------")

self.start()

cursor.execute(findaccount, (transferfrom,))

results = cursor.fetchall()

gettransferfromaccount = results[0][0]

gettransferfrombalance = results[0][1]

transferto = int(input("Enter account number to transfer to:\n"))

for row in cursor.execute(findaccount, (transferto,)):

break

else:

findaccount = ('SELECT \* FROM customerstable WHERE account\_number = ?')

cursor.execute(findaccount, (transferto,))

results = cursor.fetchall()

print(" -------------[INTERNATIONAL BANKING 5% CHARGES]----------------")

transferamount = int(input("Enter transfer amount:\n"))

charges = ((transferamount \* 5) / 100)

totalcharges = transferamount + charges

if gettransferfrombalance >= totalcharges:

transfromcurrentbalance = gettransferfrombalance - totalcharges

cursor.execute("UPDATE customers SET accbalance = %d WHERE accnumber = %d" % (transfromcurrentbalance, transferfrom))

connection.commit()

time = datetime.datetime.now()

print("-----[A/C No:", transferfrom, "successfully transferred : Rp", transferamount, "to A/C No:",transferto, "]-----")

self.records.append("%d transfered %d to account %d: at %s" % (transferfrom, transferamount, transferto, time))

print(64 \* "-")

comfirm = input("RETURN TO SUB MENU?\t\t\t\t\t\t\tYES - Y | EXIT - E\n---------------------------------------\n:").lower()

if comfirm == 'y':

self.start()

elif comfirm == 'e':

print("GOODBYE")

raise SystemExit

else:

print("INVALID OPTION")

self.start()

else:

print("YOU HAVE UNSUFFICIENT BALANCE")

self.start()

cursor.execute(findaccount,(transferto,))

results = cursor.fetchall()

gettransfertoaccount = results[0][0]

gettransfertobalance = results[0][1]

print("----------------------[DOMESTIC BANKING]------------------------")

transferamount = int(input("Enter transfer amount:\n"))

if gettransferfrombalance >= transferamount:

transfromcurrentbalance = int(gettransferfrombalance - transferamount)

transtocurrentbalance = int(gettransfertobalance + transferamount)

cursor.execute("UPDATE customers SET accbalance = %d WHERE accnumber = %d" % (transfromcurrentbalance, transferfrom))

cursor.execute("UPDATE customers SET accbalance = %d WHERE accnumber = %d" % (transtocurrentbalance, transferto))

connection.commit()

time = datetime.datetime.now()

print("--[A/C No:", transferfrom, "successfully transferred : Rp", transferamount,"to A/C No:", transferto, "]--")

self.records.append("%d transfered %d to account %d: at %s" % (transferfrom, transferamount, transferto, time))

print(64 \* "-")

comfirm = input("RETURN TO SUB MENU?\t\t\t\t\t\t\tYES - Y | EXIT - E\n----------------------------------------\n:").lower()

if comfirm == 'y':

self.start()

elif comfirm == 'e':

print("GOODBYE")

raise SystemExit

else:

print("INVALID OPTION")

self.start()

else:

print("YOU HAVE UNSUFFICIENT BALANCE")

self.start()

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

#5 FUNCTION TO FOR ACCOUNNT INFORMATION

def useraccount(self):

try:

uanumber = int(input("Enter customer's A/C\n:"))

check = "SELECT accnumber FROM customers WHERE accnumber = ?"

for accounts in cursor.execute(check,[uanumber]):

break

else:

print("-------------[ NO ACCOUNT FOUND WITH A/C NO :", uanumber, "]----------------")

self.start()

check = "SELECT accbalance FROM customers WHERE accnumber = ?"

cursor.execute(check, [uanumber])

balanceresult = cursor.fetchall()

uaresult = balanceresult[0][0]

print(20 \* "-", "A/C:", uanumber, " BALANCE STATUS", 20 \* "-")

print("\t\t\t\t Current balance is Rp:", uaresult, )

print(64 \* "-")

comfirm = input("RETURN TO SUB MENU?\t\t\t\t\t\t\tYES - Y | EXIT - E\n-------------------------------------\n:").lower()

if comfirm == 'y':

self.start()

elif comfirm == 'e':

print("GOOD-BYE")

raise SystemExit

else:

print("INVALID OPTION")

self.start()

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

#6 FUNCTION TO CLOSE ACCOUNT

def remove(self):

try:

connection = sqlite3.connect("Bank.db")

cursor = connection.cursor()

ranumber = int(input("Enter customer's A/C\n:"))

check = "SELECT accnumber FROM customers WHERE accnumber = ?"

for accounts in cursor.execute(check, [ranumber]):

break

else:

print("-------------[ NO ACCOUNT FOUND WITH A/C NO :", ranumber, "]----------------")

self.start()

cursor.execute("DELETE FROM customers WHERE accnumber = ?",(ranumber,))

connection.commit()

print(64 \* "-")

comfirm = input("RETURN TO SUB MENU?\t\t\t\t\t\t\tYES - Y | EXIT - E\n---------------------------------------\n:").lower()

if comfirm == 'y':

self.start()

elif comfirm == 'e':

print("GOOD-BYE")

raise SystemExit

else:

print("INVALID OPTION")

self.start()

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

#7 FUNCTION TO PRINT ALL RECORDS

def allrecords(self):

print (19\*"-","[ LIST ALL TRANSACTIONS ]",19\*"-")

if len(self.records)== 0:

print("\t\t\t\t\t"," Your record is empty")

self.start()

else:

for i in range (0,len(self.records)):

print(i+1,"[",self.records[i],"]",)

print(64\*"-")

try:

arr\_comfirm = input("RETURN TO SUB MENU?\t\t\t\t\t\t\tYES - Y | EXIT - E\n-----------------------------------------\n:").lower()

if arr\_comfirm == 'y':

self.start()

elif arr\_comfirm == 'e':

print("GOOD-BYE")

raise SystemExit

else:

print("INVALID OPTION")

self.start()

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN")

#8 FUNCTION TO PRINT DAILY INCOME

def dailyincome(self):

#Set the income features unavailable if income is 0

if self.income != 0:

print(22 \* "-", "PRINT DAILY INCOME", 22 \* "-")

print("\t\t\t\t\tYour daily income is :", self.income)

try:

print(64 \* "-")

di\_comfirm = input("FILTER CURRENT INCOME? \t\t\t\t\t\tYES - Y | NO - N\n------------------------------\n:").lower()

if di\_comfirm == 'y':

self.incomelist = ['Maximum Transaction', 'Lowest Transaction','Back to Menu']

for i in range(0, len(self.incomelist)):

print(i + 1, "\t", self.incomelist[i])

print(64 \* "-")

dis\_comfirm = int(input("\t\t\t\t\t\tSELECT OPTION?\n———————————————————\n:"))

if dis\_comfirm == 1:

print(24 \* "-", "MAXMUM TRANSACTION", 24 \* "-")

maximum = max(self.accountstransactions, key=self.accountstransactions.get)

print("\t\t\tYour maximum income is : [", maximum, "]", self.accountstransactions[maximum])

print(64 \* "-")

self.start()

elif dis\_comfirm == 2:

print(23 \* "-", "MINUMUM TRANSACTION", 23 \* "-")

minimum = min(self.accountstransactions, key=self.accountstransactions.get)

print("\t\t\t\t\tYour minimum income is : [", minimum, "]", self.accountstransactions[minimum],[minimum])

print(64 \* "-")

elif dis\_comfirm == 3:

self.start()

else:

print("\t\t\t\t\t\tINVALID OPTION")

self.start()

elif di\_comfirm == 'n':

self.start()

else:

print("INVALID OPTION")

self.start()

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN")

else:

print("\t\t\t\t\tYour current income is :", self.income)

self.start()

#9 FUNCTION TO TO VIEW STATISTIC

def incomestatistic(self):

try:

connection = sqlite3.connect("Bank.db")

cursor = connection.cursor()

print (19\*"-","MONTHLY INCOME STATISTIC",19\*"-")

self.submenu = ["Update Statistic & Run ","View current Statistic","Back to Menu"]

# view = ""

for i in range (len(self.submenu)):

print(i+1,"\t",self.submenu[i])

# view += " ["+str(i+1)+"] "+submenu[i]+"\t\t\t\t\t"

print(64\*"-")

#print(view)

s\_comfirm = int(input("\t\t\t\t\t\tSELECT OPTION?\n----------------------------------------------------------------\n:"))

if s\_comfirm == 1:

cursor.execute("SELECT count(\*) FROM salesrecords")

self.showlastrow = cursor.fetchall()

self.thelastrow = self.showlastrow [0][0] + 1

self.counter = "day"+str(self.thelastrow)

cursor.execute("INSERT INTO salesrecords VALUES ('%s','%d')" % (self.counter, self.income))

connection.commit()

print("Income for ",self.counter.upper(),"successfully saved")

self.statistic()

self.start()

print(" ------------------- RECORD SAVED SUCCESSFUL ------------------- ")

print(64\*"-")

sr\_comfirm = input("RETURN TO SUB MENU?\t\t\t\t\t\t\tYES - Y | EXIT - E\n----------------------------------------\n:").lower()

if sr\_comfirm == 'y':

self.start()

elif sr\_comfirm == 'e':

print("GOODBYE")

raise SystemExit

else:

print("INVALID OPTION")

self.start()

elif s\_comfirm == 2:

self.statistic()

self.start()

elif s\_comfirm == 3:

self.start()

else:

print("INVALID OPTION")

self.start()

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

#10 FUNCTION TO LOCK USER SESSION

def lockscreen(self):

self.process = True

while self.process:

try:

print("ACCOUNT LOCKED..... ENTER PASSWORD TO UNLOCKED")

self.unlogin = input("Enter ID:\n")

self.unlocklogpass = input("Enter Password:\n")

if self.unlogin == login and self.unlocklogpass == logpass:

user = Transactions(login,logpass).choose()

else:

print("NO USER | WRONG PASSWORD")

except ValueError:

print ("OOPS! INVALID INPUT.. TRY AGAIN")

#11 FUNCTION TO EXIT THE APPLICATION

def exitapplication(self):

try:

log\_comfirm = input("SAVE YOUR INCOME BEFORE LOGOUT?\t\t\t\tYES - Y | NO - N\n-----------------------------\n:").lower()

if log\_comfirm == 'y':

cursor.execute("SELECT count(\*) FROM salesrecords")

self.showlastrow = cursor.fetchall()

self.thelastrow = self.showlastrow[0][0] + 1

self.counter = "day" + str(self.thelastrow)

cursor.execute("INSERT INTO salesrecords VALUES ('%s','%d')" % (self.counter, self.income))

connection.commit()

print(64 \* "-")

print("------------- Income for ", self.counter.upper(), " successfully saved ------------")

print(64 \* "-")

print("GOODBYE")

raise SystemExit

elif log\_comfirm == 'n':

print("GOODBYE")

raise SystemExit

else:

print("INVALID OPTION")

self.start()

except ValueError:

print("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

global login

global logpass

while True:

try:

print(15\*"-","WELCOME TO MY BANKING SIMULATION",15\*"-",)

print(16\*"-","ENTER YOUR LOGIN DETAILS BELOW",16\*"-",)

login = input("Enter ID\n:")

logpass = input("Enter Password\n:")

with sqlite3.connect("Bank.db") as connection:

cursor = connection.cursor()

find\_user = ('SELECT \* FROM operators WHERE tellerid = ? AND password = ?')

cursor.execute(find\_user,[(login),(logpass)])

results = cursor.fetchall()

if results:

print(64\*"-")

print("\t\t\t\t\t\t","WELCOME",login.upper())

while True:

\_\_user = Transactions(login,logpass)

\_\_user.choose()

else:

print("\t\t\t\t No Teller account found for",login)

print(64\*"-")

teller\_comfirm = input("TRY AGAIN? \t\t\t\t\t\t\tYES - Y | NO - N\n--------------------------------------\n:").lower()

if teller\_comfirm == 'y':

pass

elif teller\_comfirm == 'n':

print("GOOD-BYE")

raise SystemExit

else:

print("GOOD-BYE")

raise SystemExit

except ValueError:

print ("OOPS! INVALID INPUT.. TRY AGAIN")

except sqlite3.OperationalError:

print("Error! Connecting to the database")

**…………………………………………………………………………………………………….**

**Statistics.py Source Code**

**…………………………………………………………………………………………………….**

import matplotlib.pyplot as plt

import sqlite3

#ESTABLISHING CONNECTION BETWEEN THE PROGROM AND SQL 'Bank.db' DATABASE

connection = sqlite3.connect('Bank.db')

cursor = connection.cursor()

cursorforrow = connection.cursor()

#EMPTY LIST TO HOLD VALUES FROM THE DATABASE TABLE

days = []

data = []

getrow = "SELECT count(\*) FROM salesrecords"

getdata = "SELECT dayincome FROM salesrecords"

cursor.execute(getdata) #GET VALUE FROM THE dayincome column IN THE salesrecords table.

cursorforrow.execute(getrow) #GET NUMBER OF ROWS IN THE salesrecords table.

incomedata = cursor.fetchall() #RETRIEVE THE dayincome values.

rowcount = cursorforrow.fetchall() #RETRIEVE THE length of the row.

num = int(rowcount[0][0])

for i in range(0, num):

days.append("Day "+str(i+1))

data.append(incomedata[i])

day = [x for x in range(len(days))]

plt.plot(data, linewidth = 5)

plt.title("S-B STATISTICS", fontsize = 24)

plt.xlabel("Records recorded", fontsize = 14)

plt.xticks(day, days)

plt.ylabel("Total daily income", fontsize = 14)

plt.tick\_params(axis = "Both", labelsize = 14)

plt.show()