**Structured Programming**

**January 2023**

|  |  |
| --- | --- |
| Name: |  |
| T-Group: |  |
| Date Submitted: |  |

**Submission to Vocareum**

|  |  |
| --- | --- |
|  | # Screenshot of datetime stamp of submission to Vocareum |

**Question 1 (5 marks)**

|  |  |
| --- | --- |
| 1 | class Guest:      def \_\_init\_\_(self, passport, name, country):          self.\_passport = passport          self.\_name = name          self.\_country = country          self.\_blacklistedReason = []      @property      def passport(self):          return self.\_passport      @property      def name(self):          return self.\_name      @property      def country(self):          return self.\_country      def isBlacklisted(self):          return bool(self.\_blacklistedReason)      def blacklist(self, dateReported, reason):          self.\_blacklistedReason.append([dateReported, reason])      def \_\_str\_\_(self):          blacklisting = ""          if self.isBlacklisted():              for date, reason in self.\_blacklistedReason:                  blacklisting += f"\n{date}: {reason}"          return f"Passport Number: {self.passport}\nName: {self.name}\nCountry: {self.\_country}{blacklisting}" |
|  | screenshots output (if any) |

**Question 2 (15 marks)**

|  |  |
| --- | --- |
| 2a) | from abc import ABC, abstractmethod  class Amenity(ABC):      def \_\_init\_\_(self, itemCode, description, price):          self.\_itemCode = itemCode          self.\_description = description          self.\_price = price      @property      def itemCode(self):          return self.\_itemCode      @property      def price(self):          return self.\_price      @abstractmethod      def getFloorArea(self):          pass      def \_\_str\_\_(self):          return "{}, {}, ${:.2f}".format(self.\_itemCode, self.\_description, self.\_price) |
| 2b) | class InRoomAmenity(Amenity):      def \_\_init\_\_(self, itemCode, description, price, floorArea):          super().\_\_init\_\_(itemCode, description, price)          self.\_floorArea = floorArea      def getFloorArea(self):          return self.\_floorArea  class SharedAmenity(Amenity):      def getFloorArea(self):          return 0 |
| 2c) | i) from AmenityClasses import SharedAmenity, InRoomAmenity  if \_\_name\_\_ == "\_\_main\_\_":      Amenities = []      Amenities.append(SharedAmenity(          "BIZ-PEP", "Per entry pass to business centre (Level 3-03)", 2.00))      Amenities.append(SharedAmenity(          "HI-TEA", "Hi-Tea buffet at Sun café (Level 1-01 2pm to 4pm)", 11.99))      Amenities.append(InRoomAmenity(          "DESK-W", "Writing desk(80cm x 55cm)", 3.99, 0.44))      Amenities.append(InRoomAmenity(          "IRON-B", "Iron and ironing board (128cm x 30cm)", 2.99, 0.4))      total\_price = 0      total\_floor\_area = 0      for amenity in Amenities:          print(amenity)          total\_price += amenity.price          total\_floor\_area += amenity.getFloorArea() |
|  | ii)  print("Total price: ${:.2f}".format(total\_price))      print("Total floor area: {:.2f} sqm".format(total\_floor\_area)) |
|  |  |

**Question 3 (20 marks)**

|  |  |
| --- | --- |
| 3a) | from ExceptionsClasses import \*  class Bed:      \_TYPE\_SIZE = {"Single": 1.73, "Super": 2.03}      def \_\_init\_\_(self, bed\_type: str, price: float, description: str):          if bed\_type not in ["Single", "Super"]:              raise BookingException("Bed type must be 'Single' or 'Super'")          self.\_type = bed\_type          self.\_price = price          self.\_description = description      @property      def price(self):          return self.\_price      @property      def floorArea(self):          return self.\_TYPE\_SIZE[self.\_type]      def \_\_str\_\_(self) -> str:          return f"{self.\_description}, ${self.\_price:.2f}" |
| 3b) | class BookingException(Exception):      pass  class MinFloorAreaException(Exception):      pass |
| 3c) | from ExceptionsClasses import \*  from AmenityClasses import SharedAmenity, InRoomAmenity, Amenity  from GuestClass import Guest  from BedClass import Bed  class Room:      \_MIN\_EXIT\_SPACE = 1.84      \_TYPE\_SIZE = {"Standard": 20.0, "Deluxe": 30.0}      def \_\_init\_\_(self, room\_type, bed, price):          if room\_type not in ["Standard", "Deluxe"]:              raise BookingException("Room type must be 'Standard' or 'Deluxe'")          self.\_type = room\_type          self.\_bed = bed          self.\_amenities = []          self.\_roomPrice = price      @property      def type(self):          return self.\_type      @property      def roomPrice(self):          return self.\_roomPrice      @property      def fullprice(self):          return self.\_roomPrice + self.\_bed.price + sum([a.price for a in self.\_amenities])      def addAmenity(self, amenity: Amenity):          amentity\_size = amenity.getFloorArea()          if sum([a.getFloorArea() for a in self.\_amenities]) + amentity\_size - self.\_TYPE\_SIZE[self.\_type] > self.\_MIN\_EXIT\_SPACE:              raise MinFloorAreaException("Not enough space for this amenity")          if amenity.itemCode in [a.itemCode for a in self.\_amenities]:              raise BookingException("Duplicate item code")      def removeAmenity(self, itemCode: str):          for amenity in self.\_amenities:              if amenity.itemCode == itemCode:                  self.\_amenities.remove(amenity)                  return          raise BookingException(              f"{itemCode}: Item code not found for room {self.\_type}")      def \_\_str\_\_(self) -> str:          room\_details = f"{self.\_type} room, ${self.\_roomPrice:.2f}"          bed\_details = str(self.\_bed)          amenities\_details = "\n".join(              [str(amenity) for amenity in self.\_amenities])          amenities\_details = amenities\_details[:-1]          full\_price = f"Full Price: ${self.fullprice:.2f}"          return f"{room\_details}\n{bed\_details}{amenities\_details}\n{full\_price}\n" |
| 3d) | i) from RoomClass import \*  # Firstly creating a deluxe room  try:      super\_single\_bed = Bed("Super", 1.5, "Room with super single bed")      deluxe\_room = Room("Deluxe", super\_single\_bed, 100)      deluxe\_room.addAmenity(InRoomAmenity(          "FRIDGE", "Mini Fridge (50L)", 4.59, 0.25))      deluxe\_room.addAmenity(InRoomAmenity(          "CHAIR", "Foldable Chair (42cm x 38cm)", 2.59, 0.16))      deluxe\_room.addAmenity(InRoomAmenity(          "DESK-W", "Writing desk (80cm x 55cm)", 3.99, 0.44))      deluxe\_room.addAmenity(InRoomAmenity(          "IRON-B", "Iron and ironing board (128cm x 30cm)", 2.99, 0.4))  except (BookingException, MinFloorAreaException) as e:      print(e)  try:      single\_bed = Bed("Single", 1.5, "Room with single bed")      standard\_room = Room("Standard", single\_bed, 100)      standard\_room.addAmenity(SharedAmenity(          "GYM-PEP", "Per entry pass to gym (Level 4-01)", 1.00))      standard\_room.addAmenity(InRoomAmenity(          "FRIDGE", "Mini Fridge (50L)", 4.59, 0.25))      standard\_room.addAmenity(SharedAmenity(          "WI-FI", "One-day Wi-Fi access", 1.00))      standard\_room.addAmenity(SharedAmenity(          "GYM-PEP", "Per entry pass to gym (Level 4-01)", 1.00))  except (BookingException, MinFloorAreaException) as e:      print(e)  print(deluxe\_room)  print(standard\_room) |
|  | ii)  try:      deluxe\_room.removeAmenity("FRIDGE")      deluxe\_room.removeAmenity("GYM-PEP")  except (BookingException, MinFloorAreaException) as e:      print(e)      print("\n")  try:      standard\_room.removeAmenity("GYM-PEP")      standard\_room.removeAmenity("GYM-PEP")  except (BookingException, MinFloorAreaException) as e:      print(e)      print("\n")  print(deluxe\_room)  print(standard\_room) |
|  |  |

**Question 4 (40 marks)**

|  |  |
| --- | --- |
| 4a) | from datetime import date, timedelta  from RoomClass import \*  from GuestClass import Guest  class Booking:      \_NEXT\_ID = 1      def \_\_init\_\_(self, guest, room, check\_in\_date, check\_out\_date):          self.\_bookingID = str(Booking.\_NEXT\_ID)          Booking.\_NEXT\_ID += 1          self.\_guest = guest          self.\_room = room          self.\_checkInDate = check\_in\_date          self.\_checkOutDate = check\_out\_date          self.\_allocatedRoomNo = None          self.\_status = "Pending"          # Ensure the guest is not blacklisted          if self.\_guest.isBlacklisted():              raise BookingException("Guest is blacklisted")          # Check-out date must be at least 1 day after check-in date          if self.\_checkOutDate <= self.\_checkInDate:              raise BookingException(                  "Check-out date must be at least 1 day after check-in date")      @property      def bookingID(self):          return self.\_bookingID      @property      def checkInDate(self):          return self.\_checkInDate      @property      def checkOutDate(self):          return self.\_checkOutDate      @property      def status(self):          return self.\_status      @status.setter      def status(self, value):          self.\_status = value      @property      def passport(self):          return self.\_guest.passport      @property      def roomType(self):          return self.\_room.type      @property      def totalPrice(self):          return self.\_room.fullprice \* ((self.\_checkOutDate - self.\_checkInDate).days)      def checkIn(self, allocatedRoomNo):          if self.\_status != "Confirmed":              raise BookingException(                  "Booking status must be Confirmed for check-in")          diff = self.\_checkOutDate - self.\_checkInDate          if diff.days <= 0:              raise BookingException(                  "Check-in must be greater than the check-in date itself")          if self.\_guest.isBlacklisted():              raise BookingException("Guest is blacklisted")          self.\_status = "Checked-In"          self.\_allocatedRoomNo = allocatedRoomNo      def \_\_str\_\_(self):          room\_str = str(self.\_room)          total\_price\_str = f"Total Price: ${self.\_room.fullprice:.2f} x {(self.\_checkOutDate - self.\_checkInDate).days} nights = ${self.totalPrice:.2f}"          return (f"Booking ID: {self.\_bookingID}\n"                  f"Passport Number: {self.\_guest.passport} Name: {self.\_guest.name}\n"                  f"Check-In/Out dates: {self.\_checkInDate.strftime('%d-%b-%Y')} / {self.\_checkOutDate.strftime('%d-%b-%Y')} "                  f"Booking Status: {self.\_status}\n"                  f"{room\_str}\n{total\_price\_str}") |
| 4b) | from datetime import datetime, timedelta  from RoomClass import \*  class Hotel:      def \_\_init\_\_(self, name, roomFilename):          self.\_name = name          self.\_guests = self.setupGuests()          self.\_amenities = self.setupAmenities()          self.\_roomAvailability = self.setupRoomAvailability(roomFilename)          self.\_bookings = {}      @property      def rooms(self):          return self.\_roomAvailability      @property      def amenities(self):          return self.\_amenities      @property      def guests(self):          return self.\_guests      def printGuests(self):          for guest in self.\_guests.values():              print(guest)          print("\n")      def printAmenities(self):          for amenity in self.\_amenities:              print(amenity)          print("\n")      def printRooms(self):          for date, roomCount in self.\_roomAvailability.items():              print(f"{date}: {roomCount}")      def printBookings(self):          for booking in self.\_bookings.values():              print(booking)      def setupGuests(self):          guests = {}          infile = open("Guests.txt", "r")          for line in infile:              pp, name, country, sami\_points = line.split(",")              guests[pp.strip()] = Guest(                  pp.strip(), name.strip(), country.strip())          infile.close()          infile = open("Blacklist.txt", "r")          for line in infile:              pp, dateReported, reason = line.split(",")              g = guests.get(pp.strip())              if g is not None:                  g.blacklist(datetime.strptime(dateReported.strip(), "%d-%b-%Y").date(),                              reason.strip())          infile.close()          return guests          def setupAmenities(self):          amenities = []          infile = open("SharedAmenity.txt", "r")          for line in infile:              itemCode, desc, price = line.split(",")              amenities.append(SharedAmenity(itemCode, desc, float(price)))          infile.close()          infile = open("InRoomAmenity.txt", "r")          for line in infile:              itemCode, desc, price, floorArea = line.split(",")              amenities.append(InRoomAmenity(itemCode, desc, float(price),                                             float(floorArea)))          infile.close()          return amenities      def setupRoomAvailability(self, filename):          roomAvailability = {}          infile = open(filename, "r")          for line in infile:              dateString, standardCount, deluxeCount = line.split(",")              thisDate = datetime.strptime(dateString, "%d-%b-%Y").date()              roomAvailability[thisDate] = [int(standardCount), int(deluxeCount)]          infile.close()          return roomAvailability      def saveRoomAvailability(self, filename):          outfile = open(filename, "w")          for k, v in self.\_roomAvailability.items():              print("{},{},{}".format(k.strftime(                  "%d-%b-%Y"), v[0], v[1]), file=outfile)          outfile.close()      def searchGuest(self, passport):          for guest in self.\_guests.values():              if guest.passport == passport:                  return guest          raise BookingException("Guest not found")      def checkRoomAvailability(self, room\_type, start, end):          diff = end - start          if diff.days < 0:              raise BookingException("Invalid date range")          cureent\_date = start          while cureent\_date <= end:              std\_count, del\_count = self.\_roomAvailability[cureent\_date][                  0], self.\_roomAvailability[cureent\_date][1]              if room\_type == "Standard":                  if std\_count == 0:                      return False              elif room\_type == "Deluxe":                  if del\_count == 0:                      return False              cureent\_date += timedelta(days=1)      def listAmenity(self):          amenities = ""          for amenity in self.\_amenities:              amenities += str(amenity) + "\n"          return amenities      def getAmenity(self, itemCode):          for amenity in self.\_amenities:              if amenity.itemCode == itemCode:                  return amenity          return None      def searchBooking(self, bookingID):          bookings = []          for booking in self.\_bookings.values():              if booking.bookingID == bookingID:                  bookings.append(booking)          return bookings      def searchBookingByPassport(self, passport):          bookings = []          for booking in self.\_bookings.values():              if booking.\_guest.passport == passport:                  bookings.append(booking)          return bookings      def SubmitBooking(self, booking):          if booking.status != "Pending":              raise BookingException("Booking is not pending")          room\_type = booking.roomType          check\_in\_date = booking.checkInDate          check\_out\_date = booking.checkOutDate          availibility = self.checkRoomAvailability(              room\_type, check\_in\_date, check\_out\_date)          if availibility == False:              raise BookingException("Room are  not available")          current\_date = check\_in\_date          while current\_date <= check\_out\_date:              std\_count, del\_count = self.\_roomAvailability[current\_date][                  0], self.\_roomAvailability[current\_date][1]              if room\_type == "Standard":                  std\_count -= 1              else:                  del\_count -= 1              self.\_roomAvailability[current\_date] = [std\_count, del\_count]              current\_date += timedelta(days=1)          booking.status = "Confirmed"          self.\_bookings[booking.bookingID] = booking      def cancelBooking(self, bookingID):          booking = self.searchBooking(bookingID)          booking = booking[0]          if booking is None:              raise BookingException("Booking not found")          if booking.status != "Confirmed":              raise BookingException(                  "Booking is either cancelled or has been checked in")          booking.status = "Cancelled"          room\_type = booking.roomType          check\_in\_date = booking.checkInDate          check\_out\_date = booking.checkOutDate          current\_date = check\_in\_date          while current\_date <= check\_out\_date:              std\_count, del\_count = self.\_roomAvailability[current\_date][                  0], self.\_roomAvailability[current\_date][1]              if room\_type == "Standard":                  std\_count += 1              else:                  del\_count += 1              self.\_roomAvailability[current\_date] = [std\_count, del\_count]              current\_date += timedelta(days=1)      def checkIn(self, bookingID, allocatedRoomNo):          for booking in self.\_bookings.values():              if booking.bookingID == bookingID:                  booking.checkIn(allocatedRoomNo)                  return |
| 4c) | from HotelClass import Hotel  from datetime import datetime  from BookingClass import \*  MyHotel = Hotel("Shining Star", "availablerooms.txt")  def add\_aminities(Room):      try:          MyHotel.printAmenities          user\_option = input("Enter the Amenity ID , you want to add:").upper()          current\_Amenity = MyHotel.getAmenity(user\_option)          if (current\_Amenity == None):              raise BookingException("There exists such no item code ")          else:              Room.addAmenity(current\_Amenity)      except BookingException as e:          print(e)          add\_aminities(Room)  def search\_booking():      try:          option = input("Do you want to search by (I)D or (P)assport: ")          if option.upper() == "I":              booking\_id = input("Enter booking ID: ")              booking = MyHotel.searchBooking(booking\_id)          else:              passport = input("Enter passport number: ")              booking = MyHotel.searchBookingByPassport(passport)          if booking == None:              raise BookingException("No booking found")          print("Booking found:\n")          for bookings in booking:              print(bookings)      except ValueError:          print("Invalid booking ID")          print("\n")          search\_booking()      except BookingException as e:          print(e)          print("\n")          search\_booking()  def checkin\_hotel():      try:          booking\_id = input("Enter booking ID: ")          Booking = MyHotel.searchBooking(booking\_id)          if Booking == None:              raise BookingException("No booking found")          print(Booking[0])          room = input("Enter room number:  or e to exit:")          if room.upper() == "E":              print("Check in cancelled")              return          MyHotel.checkIn(booking\_id, room)          print("Check in successful")      except ValueError:          print("Invalid booking ID")          checkin\_hotel()      except BookingException as e:          print(e)          checkin\_hotel()  def cancel\_booking():      try:          selected\_id = input("Enter booking ID: or e to exit to main menu: ")          if selected\_id.upper() == "E":              return          booking = MyHotel.searchBooking(selected\_id)          booking = booking[0]          if booking == None:              raise BookingException("No booking found")          MyHotel.cancelBooking(selected\_id)          print("Booking cancelled")      except BookingException as e:          print(e)          print("\n")          cancel\_booking()      except ValueError:          print("Invalid booking ID")          print("\n")          cancel\_booking()  def select\_room():      try:          room\_type = input(              "Enter room type: (S) for Standard, (D) for Deluxe: ")          if room\_type.upper() == "S":              room\_type = "Standard"              room\_price = 23          elif room\_type.upper() == "D":              room\_type = "Deluxe"              room\_price = 19          else:              raise BookingException("Please enter S or D , To select room type")          return room\_type, room\_price      except BookingException as e:          print(e)          print("\n")          select\_room()  def select\_bed():      try:          bed\_type = input(              "Enter bed type: (S) for Single, (D) for Super Single Bed: ")          if bed\_type.upper() == "S":              bed\_type = "Single"              price = 4.73              description = "Single bed"          elif bed\_type.upper() == "D":              bed\_type = "Super"              price = 3.33              description = "Super Single Bed"          else:              raise BookingException(                  "Please enter S or D to select the Bed of your choice")          return bed\_type, price, description      except BookingException as e:          print(e)          print("\n")          select\_bed()  def select\_guest():      try:          guest\_id = (input("Enter PassPort Number: or e to exit to main menu: "))          if guest\_id.upper() == "E":              return          guest = MyHotel.searchGuest((guest\_id))          print("Guest selected:")          print(guest)          return guest      except ValueError:          print("Invalid guest ID")          print("\n")          select\_guest()      except BookingException as e:          print(e)          print("\n")          select\_guest()  def doBooking(MyHotel):      guest = select\_guest()      if guest == None:          return      try:          room\_type, room\_price = select\_room()          bed\_type, bed\_price, bed\_description = select\_bed()          current\_bed = Bed(bed\_type, bed\_price, bed\_description)          current\_room = Room(room\_type, current\_bed, room\_price)          print("Room selected:")          print(current\_room)          input("Press enter to continue")          op = input("Do you want to add amenities? (Y/N): ")          if op.upper() == "N":              pass          else:              while True:                  print("Amenities available:")                  MyHotel.printAmenities()                  add\_aminities(current\_room)                  user\_option = input(                      "Do you want to add more amenities? (Y/N): ")                  if user\_option.upper() == "N":                      break          check\_in\_date = input("\nEnter check-in date DD-MON-YY: ")          check\_out\_date = input("Enter check-out date DD-MON-YY: ")          check\_in\_date = datetime.strptime(check\_in\_date, "%d-%b-%Y").date()          check\_out\_date = datetime.strptime(check\_out\_date, "%d-%b-%Y").date()          CurrentBooking = Booking(              guest, current\_room, check\_in\_date, check\_out\_date)          MyHotel.SubmitBooking(CurrentBooking)          print("Booking successful")      except BookingException as e:          print(e)          print("Booking failed")          print("\n")          doBooking(MyHotel)  def MainMenu():      print("@@SAMI HOTEL MANAGEMENT SYSTEM@@")      print("1. Submit Booking")      print("2. Cancel Booking")      print("3. Search Booking")      print("4. Check-in")      print("5. Exit")      option = input("Enter option: ")      if option == "1":          doBooking(MyHotel)      elif option == "2":          cancel\_booking()      elif option == "3":          search\_booking()      elif option == "4":          checkin\_hotel()      elif option == "5":          MyHotel.saveRoomAvailability("availablerooms.txt")          print("Thank you for using the system")          return      else:          print("Invalid option")          MainMenu()      if option != "5":          print("\n")          input("Press enter to continue")          MainMenu()  MainMenu() |
|  |  |

**Question 5 (20 marks)**

|  |  |
| --- | --- |
| 5a)  5b) | import tkinter as tk  from HotelClass import Hotel  from BookingClass import \*  import date\_validator  class MyGUI:      def \_\_init\_\_(self, master):          self.master = master          master.title("SAMI Blacklist GUI-done by XXXX")          self.myHotel = Hotel("SAMI", "availablerooms.txt")          # Create the labels and their text fields          self.passport\_label = tk.Label(master, text="Passport:")          self.passportentry = tk.Entry(master)          self.date\_label = tk.Label(master, text="Date Reported:")          self.date\_entry = tk.Entry(master)          self.date\_entry.config(state='disabled')          self.reason\_laebel = tk.Label(master, text="Reason(s):")          self.reason\_entry = tk.Entry(master)          # disable entry3          self.reason\_entry.config(state='disabled')          # date format          self.label4 = tk.Label(master, text="(dd-mon-yyyy)")          # Create the buttons          self.search = tk.Button(master, text="Search", command=self.search\_)          self.blacklist = tk.Button(              master, text="Blacklist", command=self.blacklist\_)          # disable button2          self.blacklist.config(state='disabled')          self.reset = tk.Button(master, text="Reset", command=self.reset\_)          # Create the large text field          self.text = tk.Text(master)          # Arrange the widgets in the grid          self.passport\_label.grid(row=0, column=0, padx=5, pady=5)          self.passportentry.grid(row=0, column=1, padx=5, pady=5)          self.date\_label.grid(row=1, column=0, padx=5, pady=5)          self.date\_entry.grid(row=1, column=1, padx=5, pady=5)          self.reason\_laebel.grid(row=2, column=0, padx=5, pady=5)          self.reason\_entry.grid(row=2, column=1, padx=5, pady=5)          self.label4.grid(row=1, column=2, padx=5, pady=5)          self.search.grid(row=3, column=0, padx=5, pady=5)          self.blacklist.grid(row=3, column=1, padx=5, pady=5)          self.reset.grid(row=3, column=2, padx=5, pady=5)          self.text.grid(row=4, column=0, columnspan=3, padx=5, pady=5)      def search\_(self):          # enable entry2          self.date\_entry.config(state='normal')          # enable entry3          self.reason\_entry.config(state='normal')          # enable button2          self.blacklist.config(state='normal')          # disable button 1          self.search.config(state='disabled')          # disable entry1          self.passportentry.config(state='disabled')          # get the passport number          self.passport = self.passportentry.get()          # search for the guest          try:              self.guest = self.myHotel.searchGuest(self.passport)              # check if guest is blacklisted              if self.guest.isBlacklisted():                  # if guest is blacklisted                  # display the guest's details                  self.text.insert(tk.END, str(self.guest))                  self.text.insert(tk.END, "\nGuest is already blacklisted")                  # disable blackilst button,reason and date                  self.blacklist.config(state='disabled')                  self.date\_entry.config(state='disabled')                  self.reason\_entry.config(state='disabled')              else:                  # if guest is not blacklisted                  # display the guest's details                  self.text.insert(tk.END, str(self.guest))                  # display the date and reason fields                  self.text.insert(                      tk.END, "\nEnter date and reason to blacklist: \n")          except:              self.text.insert(tk.END, "\nGuest not found!")      def blacklist\_(self):          # get date          self.date = self.date\_entry.get()          # get reason          self.reason = self.reason\_entry.get()          # check if date is valid and also error message if reason is not entered          if date\_validator.validate\_date(self.date) and self.reason != "":              # blacklist the guest              self.guest.blacklist(self.date, self.reason)              # display the guest's details              self.text.insert(tk.END, str(self.guest))              # disable blackilst button,reason and date              self.blacklist.config(state='disabled')              self.date\_entry.config(state='disabled')              self.reason\_entry.config(state='disabled')              # open Blacklist.txt for appending              self.file = open("Blacklist.txt", "a")              # write the guest's details to the file PA0010023, 12-Mar-2023, Causes damage to facilities in this format              self.file.write("\n"+self.passportentry.get() +                              ", " + self.date + ", " + self.reason+"\n")              self.file.close()              # display message that guest is blacklisted on date due to              self.text.insert(tk.END, "\n<<blacklisted on date, due to>> ")              # display date,reason              self.text.insert(tk.END, "\n"+self.date + ", " + self.reason)          else:              # if date is not valid              if date\_validator.validate\_date(self.date) == False:                  self.text.insert(tk.END, "\nInvalid date format!")              # if reason is not entered              if self.reason == "":                  self.text.insert(tk.END, "\nReason not entered!")              # if date is empty              if self.date == "":                  self.text.insert(tk.END, "\nDate not entered!")      def reset\_(self):          # enable all entries          self.passportentry.config(state='normal')          self.date\_entry.config(state='normal')          self.reason\_entry.config(state='normal')            # empty al text fields          self.passportentry.delete(0, tk.END)          self.date\_entry.delete(0, tk.END)          self.reason\_entry.delete(0, tk.END)          self.text.delete(1.0, tk.END)          # enable entry1          self.passportentry.config(state='normal')          # disable entry2          self.date\_entry.config(state='disabled')          # disable entry3          self.reason\_entry.config(state='disabled')          # enable button1          self.search.config(state='normal')          # disable button2          self.blacklist.config(state='disabled')    root = tk.Tk()  my\_gui = MyGUI(root)  root.mainloop() |
|  |  |