

|  |
| --- |
| Assignment #  Winter-2025 |
|  |
| March 2  Course Title: Programming Principles  Course Code: PROG10004  Authored by:  Student Name: Muhammad Mukry Student Number: 991798855 |

Contents

[Assignment 3 4](#_Toc191859642)

[Question #1: 5](#_Toc191859643)

[UML Diagram: 6](#_Toc191859644)

[Python Code: 6](#_Toc191859645)

[Flowchart (if applicable): 10](#_Toc191859646)

[Repository (if applicable): 11](#_Toc191859647)

[Question 2: 12](#_Toc191859648)

[Python Code: 12](#_Toc191859649)

[Flowchart (if applicable): 14](#_Toc191859650)

[Repository (if applicable): 15](#_Toc191859651)

[Question #3: 15](#_Toc191859652)

[Python Code: 15](#_Toc191859653)

[Flowchart (if applicable): 17](#_Toc191859654)

[Repository (if applicable): 18](#_Toc191859655)

[Question 4: 18](#_Toc191859656)

[UML Diagram: 20](#_Toc191859657)

[Flowchart (if applicable): 21](#_Toc191859658)

[Python Code: 21](#_Toc191859659)

[Limitations and Challenges 34](#_Toc191859660)

# Assignment 3

|  |
| --- |
| Question #1: 1. You are required to design a Game Simulation using Object-Oriented Programming (OOP) in Python. The system consists of three main classes: Application, Game, and Player.  Application Class  Contains an instance of the Game class.  Responsible for managing and running the game for 5 rounds.  Game Class  Attributes:  players: A list containing three Player objects.  guess: A randomly generated number between 1 and 6 (initialized at the start of each round).  Methods:  start(): Initializes the guess attribute with a random integer between 1 and 6.  match(): Compares the guess with each player’s roll. If a player's roll matches the guess, their score is incremented.  Player Class  Attributes:  score: Stores the player’s current score.  Methods:  \_\_init\_\_(): Initializes the player’s score to 0.  roll(): Returns a random integer between 1 and 6 (simulating a dice roll).  Game Rules & Flow:  The Application class creates a Game instance and starts the game.  In each round:  The start() method sets a random guess.  Each player rolls a number between 1 and 6.  The match() method compares the guess with each player's roll, updating the score accordingly.  The game runs for 5 rounds.  At the end of the game, the player with the highest score is declared the winner.  Tasks:  Construct a UML Diagram to illustrate class relationships.  Sketch a Flowchart to visualize the process.  Implement the Python Code for all classes.  Provide screenshots of the output demonstrating the gameplay.   UML Diagram:  Python Code: Application.py  from Game import Game  class Application:      def play\_game(self, number\_of\_game\_rounds):           my\_Game = Game(number\_of\_game\_rounds)           my\_Game.start()      def run(self):  *#Starting the game to run for 5 rounds*          self.play\_game(5)    if \_\_name\_\_ == "\_\_main\_\_":      app = Application()      app.run()  Game.py  from Player import Player  import random  class Game:      Player\_list = [Player("Muhammad"), Player("Mustafa"), Player("Ayesha")]      def \_\_init\_\_(self, number\_of\_game\_rounds):          self.number\_of\_game\_rounds = number\_of\_game\_rounds          self.guess = 0  *#Compares the guess with each player’s roll. If a player's roll matches the guess, their score is incremented.*      def match(self, player\_dice\_value, player\_number):          if self.guess == player\_dice\_value:              self.Player\_list[player\_number].score += 1              print(f'{self.Player\_list[player\_number].player\_name} score is {self.Player\_list[player\_number].score}')        def start(self):  *#Running the number of rounds for the game*              for i in range(self.number\_of\_game\_rounds):  *#Generating the guess value for each round*                  self.guess = random.randint(1, 6)                  print(f'Round {i} has Guess value of: {self.guess}')  *#Rolling the dice for each player*                  for player\_number in range (len(self.Player\_list)):                      player\_dice\_value = self.Player\_list[player\_number].roll()                      print(f'The dice value for {self.Player\_list[player\_number].player\_name} is {player\_dice\_value}')  *#Calling the match method to compare if the Player dice value is the same as guess value*                      self.match(player\_dice\_value, player\_number)  *#Printing the finalscore for each player*              for player\_number in range (len(self.Player\_list)):                  print(f'{self.Player\_list[player\_number].player\_name} final score is {self.Player\_list[player\_number].score}')  *#If Player 1 is the winner*              if  (self.Player\_list[0].score >  self.Player\_list[1].score) and  (self.Player\_list[0].score >  self.Player\_list[2].score):                   print(f'{self.Player\_list[0].player\_name} with final score of {self.Player\_list[0].score} is the winner')  *#If Player 2 is the winner*              if  (self.Player\_list[1].score >  self.Player\_list[0].score) and  (self.Player\_list[1].score >  self.Player\_list[2].score):                   print(f'{self.Player\_list[1].player\_name} with final score of {self.Player\_list[1].score} is the winner')  *#If Player 3 is the winner*              if  (self.Player\_list[2].score >  self.Player\_list[0].score) and  (self.Player\_list[2].score >  self.Player\_list[1].score):                   print(f'{self.Player\_list[2].player\_name} with final score of {self.Player\_list[2].score} is the winner')  *#If Player 1 and Player 2 have a tie*              if  (self.Player\_list[0].score ==  self.Player\_list[1].score) and  (self.Player\_list[0].score >  self.Player\_list[2].score):                   print(f'{self.Player\_list[0].player\_name} and {self.Player\_list[1].player\_name} with final score of {self.Player\_list[0].score} has a tie')    *#If Player 1 and Player 3 have a tie*              if  (self.Player\_list[0].score ==  self.Player\_list[2].score) and  (self.Player\_list[0].score >  self.Player\_list[1].score):                   print(f'{self.Player\_list[0].player\_name} and {self.Player\_list[2].player\_name} with final score of {self.Player\_list[2].score} has a tie')  *#If Player 2 and Player 3 have a tie*              if  (self.Player\_list[1].score ==  self.Player\_list[2].score) and  (self.Player\_list[1].score >  self.Player\_list[0].score):                   print(f'{self.Player\_list[1].player\_name} and {self.Player\_list[2].player\_name} with final score of {self.Player\_list[2].score} has a tie')  *#If Player 1 and Player 2 and Player 3 have a tie*              if  (self.Player\_list[0].score ==  self.Player\_list[1].score) and  (self.Player\_list[0].score ==  self.Player\_list[2].score):                   print(f'{self.Player\_list[0].player\_name}, {self.Player\_list[1].player\_name} and {self.Player\_list[2].player\_name} with final score of {self.Player\_list[2].score} has a tie')  Player.py  import random  class Player:      def \_\_init\_\_(self, player\_name):          self.player\_name = player\_name          self.score = 0      def roll(self):          return random.randint(1,6)    Flowchart (if applicable):   Paste the screenshot of your output here   Repository (if applicable): State your git repository and give a screenshot of the directory contents (if applicable) |
| Question 2: 2. Design a program that generates an optical illusion using squares of varying sizes and colors. The squares should be arranged in such a way that they create the visual impression of a 3D cube.  Task Requirements:  Implement a Python program to generate the optical illusion.  Name the file Shapev1.py and store it as Version 1 in your Git repository.  Capture and provide the following:  A screenshot of your code.  A screenshot of the program’s output.  A flowchart illustrating the logic of your program. Python Code: import turtle  #Method to draw square  def draw\_square(my\_turtle, line\_color, fill\_color, square\_size, rotation\_angle):      my\_turtle.color(line\_color, fill\_color)      my\_turtle.begin\_fill()      my\_turtle.forward(square\_size)      my\_turtle.left(rotation\_angle)      my\_turtle.forward(square\_size)      my\_turtle.left(rotation\_angle)      my\_turtle.forward(square\_size)      my\_turtle.left(rotation\_angle)      my\_turtle.forward(square\_size)      my\_turtle.end\_fill()  #Setting up the turtle screen  wn = turtle.Screen()  my\_turtle = turtle.Turtle()  my\_turtle.shape('classic')  #Printing squares of different sizes to show a cube feeling  for i in range(24):      if i % 2 == 0:          my\_turtle.penup()          my\_turtle.left(90)          my\_turtle.forward(125 + (i \* 2))          my\_turtle.left(90)          my\_turtle.forward(125 + (i \* 2))          my\_turtle.pendown()          draw\_square(my\_turtle,"red", "red", 100, 90)      else:            my\_turtle.penup()          my\_turtle.left(90)          my\_turtle.forward(125)          my\_turtle.left(90)          my\_turtle.forward(125)          my\_turtle.pendown()          draw\_square(my\_turtle,"yellow", "yellow", 75, 90)    wn.exitonclick() Flowchart (if applicable):   Paste the screenshot of your output here   Repository (if applicable): State your git repository and give a screenshot of the directory contents (if applicable) Question #3: 3. Using the logic of Question2, develop an optical illusion of sphere. Name this file as Shapev2 and store it as version 2 in your git repository. Provide screenshot of code, outputs, as well as a flowchart for your program Python Code: Shapev2.py  import turtle *#Importing turtle*  *#Method to draw sphere*  def draw\_circle(my\_turtle, line\_color, fill\_color, circle\_radius, value\_to\_decrease\_circle\_size):      my\_turtle.color(line\_color, fill\_color)      my\_turtle.begin\_fill()      my\_turtle.circle(circle\_radius - (value\_to\_decrease\_circle\_size))      my\_turtle.end\_fill()  *#Setting up the turtle screen*  wn = turtle.Screen()  my\_turtle = turtle.Turtle()  my\_turtle.shape()  *#Printing circles of two different sizes to show a sphere feeling*  for i in range(50):      if i % 2 == 0:          my\_turtle.penup()          my\_turtle.left(90)          my\_turtle.forward(150)          my\_turtle.left(90)*#*          my\_turtle.forward(150)          my\_turtle.pendown()          if i % 20  == 0:              draw\_circle(my\_turtle,"#FFFFCB", "#FFFFCB", 100, i)          else:              draw\_circle(my\_turtle,"#FFFF01", "#FFFF01", 100, i)      else:          my\_turtle.penup()          my\_turtle.left(90)          my\_turtle.forward(150)          my\_turtle.left(90)*#*          my\_turtle.forward(150)          my\_turtle.pendown()          if i % 3 == 0 or i % 5  or i % 7== 0:              draw\_circle(my\_turtle,"#FF4500", "#FF4500", 50, i)          else:              draw\_circle(my\_turtle,"#FF7F50", "#FF7F50", 50, i)  my\_turtle.shape()  wn.exitonclick() Flowchart (if applicable):   Paste the screenshot of your output here   Repository (if applicable): State your git repository and give a screenshot of the directory contents (if applicable) Question 4: |
| 1. You are required to implement an Account Management System using Object-Oriented Programming (OOP) principles in Python. The system will involve three primary classes: Account, Checking, and Savings, along with an additional Person class that interacts with the account.   Account (Parent Class)  Attributes:  Acct\_type: Represents the type of account (e.g., "Checking" or "Savings").  min\_balance: A minimum balance requirement (default: 500).  current\_balance: Stores the current account balance.  Methods:  withdraw(amount): Decrements current\_balance by amount. If the balance goes negative, it should check if the overdraft limit is exceeded.  deposit(amount): Increments current\_balance by amount.  Checking (Child Class of Account)  Additional Attributes:  over\_draft: Set to 1000 (overdraft limit).  Inherits the withdraw and deposit methods.  Savings (Child Class of Account)  Additional Attributes:  over\_draft: Set to 1200 (overdraft limit).  Additional Methods:  profit(): Increments current\_balance by 15% (profit is credited periodically).  Person Class  Attributes:  name: Stores the name of the person.  account: An instance of either Checking or Savings.  Functionality:  The user can create an instance of Person and select an account type.  The user can perform deposits and withdrawals interactively. If current\_balance goes negative, overdraft is increased by 200, and a warning is displayed.  Every 10 deposits, the profit() method of Savings (if applicable) is called.  Tasks:   * Construct a UML Diagram to represent the class relationships. * Sketch a Flowchart to visualize the workflow. * Implement the Python Code for the system. * Show sample outputs demonstrating various functionalities. * Identify limitations and challenges in the implementation. |

## UML Diagram:

A screenshot of a computer

AI-generated content may be incorrect.

## Flowchart (if applicable):

A diagram of a customer

AI-generated content may be incorrect.

## Python Code:

Application.py

from Account import Account

from Checking import Checking

from Savings import Savings

from Person import Person

class Application:

*#Initializing the empty customer/person list*

    customer\_list = []

*#Method to display main menu for customer*

    def display\_main\_menu(self):

        user\_choice = 0

        while user\_choice != 3:

            print("\*\*\*\*\*\*\*\*\*\* Main Menu for Customer \*\*\*\*\*\*\*\*\*\*")

            print("Press 1. To create new customer: ")

            print("Press 2. To select a customer from a list ")

            print("Press 3. To Exit the program ")

            user\_choice = int(input("Enter your choice(1 or 2 or 3 ): "))

*#User chooses to create a new customer/person*

            if user\_choice == 1:

                person = self.create\_new\_customer()

*#If user successfully created a new customer/person then offer them options*

*# to perform to deposit/withdraw transactions*

                if person is not None:

                    self.display\_sub\_menu\_account\_activities(person)

*#User chooses to select an existing customer/person*

            elif user\_choice == 2:

                person = self.list\_customer()

*#If user selects a valid existing customer/person then offer them options*

*# to perform to deposit/withdraw transactions*

                if person is not None:

                    self.display\_sub\_menu\_account\_activities(person)

*#Method to display sub menu for customer/person to perform deposit/withdraw transactions*

    def display\_sub\_menu\_account\_activities(self, person):

        user\_choice = 0

        while user\_choice != 3:

            print("\*\*\*\*\*\*\*\*\*\* Sub Menu for Transactions \*\*\*\*\*\*\*\*\*\*")

            print("Press 1. To Deposit amount: ")

            print("Press 2. To Withdraw amount: ")

            print("Press 3. To Exit back to Main Menu")

            user\_choice = int(input("Enter your choice(1 or 2 or 3 ): "))

*#If user choose to deposit amount*

            if user\_choice == 1:

                deposit\_amount = int(input("Enter amount to deposit: "))

                person.account.deposit(deposit\_amount)

*#Condition to check if customer is eligible for profit*

                if (person.account.account\_type == "Savings") and (person.account.deposit\_counter == 10):

                     person.account.profit()

                     person.account.deposit\_counter = 0

*#If user choose to withdraw amount*

            elif user\_choice == 2:

                withdraw\_amount = int(input("Enter amount to withdraw: "))

                person.account.withdraw(withdraw\_amount)

*#Method to create new customer/person and add them to customer/person list*

    def create\_new\_customer(self):

        account = None

        customer\_name = input("Enter customer name: ")

        account\_type = input("Type of account to open(Savings/Checking): ")

*#If user selects Savings account type then create Savings account with the*

*# default minimum balance and overdraft limit for Savings account*

        if account\_type == "Savings":

            account = Savings("Savings", 500, 0, 1200)

*#If user selects Checking account type then create Checking account with the*

*# default minimum balance and overdraft limit for Checking account*

        elif account\_type == "Checking":

            account = Checking("Checking", 500, 0, 1000)

*#If user selected the correct account types and Account is successfully created*

*# then create a new customer with the provided name and account type*

        if account is not None:

            person = Person(customer\_name, account)

            self.customer\_list.append(person)

            return person

*#Method to list existing customers/persons and select customer /person from the list*

    def list\_customer(self):

        for person in self.customer\_list:

            print(f'Person| account type | min balance | current balance | overdraft limit')

            print(f'{person.name} |  {person.account.account\_type} | {person.account.min\_balance} | {person.account.current\_balance} | {person.account.overdraft\_limit} |')

*#If any customer exists then request the option to select customer*

        if len(self.customer\_list) > 0:

            selected\_customer\_name = input("Enter customer to select: ")

            for person in self.customer\_list:

                if person.name == selected\_customer\_name:

                    return  person

*# Else notify that no customer exists*

        else:

            print(f'No customers exist!')

*#Method to run the main menu options*

    def run(self):

        self.display\_main\_menu()

if \_\_name\_\_ == "\_\_main\_\_":

    app = Application()

    app.run()

Account.py

class Account:

*#defining Account attributes*

    account\_type = ""

    min\_balance = 0

    current\_balance = 0

    overdraft\_limit= 0

    deposit\_counter = 0

    def \_\_init\_\_(self, account\_type, min\_balance, current\_balance,overdraft\_limit):

        self.account\_type = account\_type

        self.min\_balance = min\_balance

        self.current\_balance = current\_balance

        self.overdraft\_limit= overdraft\_limit

*#Method to withdraw amount and also check if it is within overdraft limit*

    def withdraw(self, withdraw\_amount):

*#If withdrawal amount is less than current balance then decrease current total balance*

*# by subtracting withdrawal amount*

        if (self.current\_balance - withdraw\_amount)>= 0:

            self.current\_balance -= withdraw\_amount

            print(f'The current balance after withdrawal is: {self.current\_balance}')

        else:

*#If the withdrawal amount results in current total amount greater than overdraft*

*# then reject with a warning message*

            if (self.current\_balance - withdraw\_amount) \* -1 > self.overdraft\_limit:

                print(f'Overdraft limit exceeded: Cannot withdraw requested amount of ${withdraw\_amount}')

*#Else if the withdrawal amount results in current total amount less than overdraft*

*# but less than 0 then withdraw the amount and update current total with a warning message*

*# and also increase overdraft limit by $200*

            else:

                self.overdraft\_limit += 200

                self.current\_balance -= withdraw\_amount

                print(f'Warning: You are using overdraft limit which is now increased to ${self.overdraft\_limit}')

                print(f'Warning: Your current balance is: ${self.current\_balance} ')

*#Method to deposit the amount and increase the deposit counter by 1*

    def deposit(self, deposit\_amount):

            self.deposit\_counter += 1

            self.current\_balance += deposit\_amount

            print(f'The current balance after deposit is: {self.current\_balance}')

Savings.py

from Account import Account

*#Creating a child class Savings inheriting from class Account*

class Savings(Account):

    def \_\_init\_\_(self, account\_type, min\_balance, current\_balance,overdraft\_limit):

*#Executing the constructor for the parent class*

        super().\_\_init\_\_(account\_type, min\_balance, current\_balance,overdraft\_limit)

*#Method to create the profit*

    def profit(self):

        self.current\_balance += (self.current\_balance \* 0.15)

        print(f'The current balance after profit is: {self.current\_balance}')

Checking.py

from Account import Account

*#Creating a child class Checking inheriting from class Account*

class Checking(Account):

    def \_\_init\_\_(self, account\_type, min\_balance, current\_balance,overdraft\_limit):

*#Creating a child class Checking inheriting from class Account*

        super().\_\_init\_\_(account\_type, min\_balance, current\_balance,overdraft\_limit)

Person.py

*#Person Class*

*#Attributes:*

*#name: Stores the name of the person.*

*#account: An instance of either Checking or Savings.*

from Account import Account

class Person:

*#defining Person attributes*

    name = ""

    account = Account

*#Constructor to create Person object*

    def \_\_init\_\_(self, name, account):

        self.name = name

        self.account = account

Paste the screenshot of your output here

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

Successfully calculated the profit at the 10th deposit

A screenshot of a computer screen

AI-generated content may be incorrect.

Successfully withdrawn requested amount from the account

A screenshot of a computer program

AI-generated content may be incorrect.

Successfully withdrawn requested amount using the overdraft limit and also increasing the overdraft limit by additional $200 for the Savings account

A screenshot of a computer screen

AI-generated content may be incorrect.

Rejected the withdrawal amount request exceeding the overdraft limit with warning message for the Savings account.

A screenshot of a computer screen

AI-generated content may be incorrect.

Successfully created a new customer with checking account

Added $100 to the Checking account

Withdrawn $56 from the Checking account

A screenshot of a computer program

AI-generated content may be incorrect.

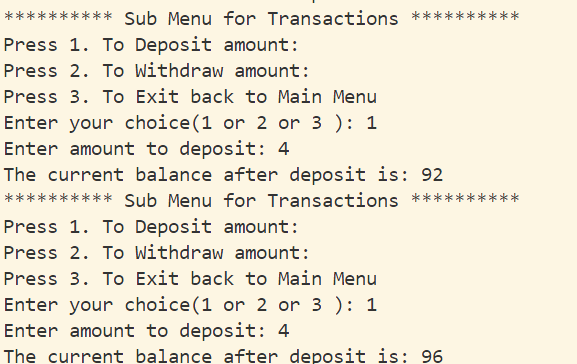
Showing all the customers with their account details and option to select the customer to perform transactions for.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.



Successfully selected customer with Checking account type from the list of existing customers.

Performed more than 10 deposits and no profit is calculated as this is Checking account.

A screenshot of a computer

AI-generated content may be incorrect.

Successfully withdrawn requested amount using the overdraft limit and also increasing the overdraft limit by additional $200 for the Checking account.

Rejected the withdrawal amount request exceeding the overdraft limit with warning message for the Checking account.

## Limitations and Challenges

1. A customer can have only one type of account
2. Overdraft limit could increase to a really high limit if customer keep withdrawing under the overdraft and keeps getting increased in overdraft limit by $200 each time