**Question 1:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Question 2:**

**Code:**

import random

import numpy as np

import pandas as pd

import os

class DataGenerator:

def \_\_init\_\_(self, n\_rows=500, save\_path="data/raw/generated\_data.csv", seed=None):

self.n\_rows = n\_rows

self.save\_path = save\_path

if seed is not None:

random.seed(seed)

np.random.seed(seed)

def generate\_numeric\_features(self):

"""Generate numerical features such as age, income, etc."""

age = np.random.randint(18, 70, self.n\_rows)

income = np.random.randint(20000, 150000, self.n\_rows)

score = np.random.randint(0, 100, self.n\_rows)

years\_of\_experience = np.random.randint(0, 40, self.n\_rows)

purchases\_last\_year = np.random.randint(0, 50, self.n\_rows)

return age, income, score, years\_of\_experience, purchases\_last\_year

def generate\_categorical\_features(self):

"""Generate categorical features such as gender, product type, etc."""

gender = np.random.choice(["Male", "Female", "Other"], self.n\_rows, p=[0.48, 0.48, 0.04])

product\_type = np.random.choice(["Electronics", "Clothing", "Food", "Books"], self.n\_rows)

return gender, product\_type

def generate\_target(self, income, score):

"""Generate a binary target (classification)."""

# Higher income and score → more likely to purchase

probs = (income / income.max()) \* 0.5 + (score / 100) \* 0.5

purchase = np.random.binomial(1, probs)

return purchase

def generate\_dataset(self):

"""Generate the entire dataset and return as a DataFrame."""

age, income, score, exp, purchases = self.generate\_numeric\_features()

gender, product = self.generate\_categorical\_features()

purchase = self.generate\_target(income, score)

data = pd.DataFrame({

"Age": age,

"Income": income,

"Score": score,

"Experience": exp,

"Purchases\_Last\_Year": purchases,

"Gender": gender,

"Product\_Type": product,

"Purchased": purchase

})

return data

def save\_dataset(self, data):

"""Save dataset to CSV."""

os.makedirs(os.path.dirname(self.save\_path), exist\_ok=True)

data.to\_csv(self.save\_path, index=False)

print(f" Dataset saved to {self.save\_path}")

def run(self):

"""Generate and save dataset in one go."""

data = self.generate\_dataset()

self.save\_dataset(data)

return data

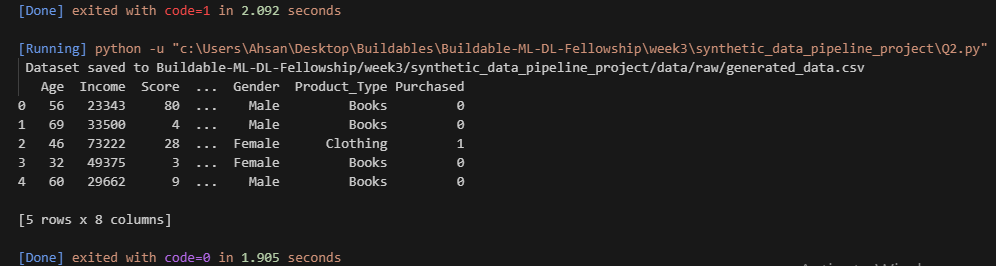
# Example usage

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

generator = DataGenerator(n\_rows=500, save\_path="data/raw/generated\_data.csv", seed=42)

df = generator.run()

print(df.head())

****

**Report:**

**Object-Oriented Programming (OOP) and Reusability in Data Generator**

**Introduction**

In this task, a Python class DataGenerator was developed to create synthetic datasets that mimic real-world scenarios. The generator produces numerical features (e.g., age, income), categorical features (e.g., gender, product type), and a binary target column (e.g., purchase decision). The dataset is saved to a CSV file for later use.

This report explains how **Object-Oriented Programming (OOP)** principles and **reusability** contribute to making the generator flexible, scalable, and easy to maintain.

**Role of OOP in the Data Generator**

**1. Encapsulation**

All dataset creation logic is bundled inside the DataGenerator class. This ensures that related methods—such as generating numerical features, categorical features, target variables, and saving data—are organized in one place. As a result, the code becomes easier to understand, debug, and maintain.

**2. Abstraction**

Each function within the class performs a clear and specific role (e.g., generate\_numeric\_features, generate\_categorical\_features, generate\_target). This abstraction hides complex implementation details from the user. The user only needs to call the run() method without worrying about the underlying data generation process.

**3. Reusability**

Because the class is modular, the same codebase can be reused for multiple applications. For example:

* In a **customer dataset**, features might include income and purchases.
* In a **patient dataset**, features could include age, blood pressure, and disease status.  
  Only minor modifications are required, while the overall structure remains reusable.

**4. Scalability and Flexibility**

The constructor (\_\_init\_\_) allows the user to adjust key parameters such as number of rows, file save path, and random seed. This flexibility makes it easy to scale up to larger datasets, change output locations, or ensure reproducibility without rewriting the code.

**5. Maintainability**

If an error or improvement is needed—for example, refining the probability distribution for the target variable—the fix only needs to be made in one method. This centralized approach reduces duplication and makes the code more maintainable over time.

**Reusability Benefits in Practice**

1. **Expanding Features**  
   New features such as "Education Level," "Marital Status," or "Transaction Type" can be added by creating additional methods. Existing code does not need to be rewritten.
2. **Different Scenarios**  
   By changing just the generation logic, the same class can produce synthetic data for customers, students, patients, or financial transactions.
3. **Task Adaptability**  
   The class can be extended to support **regression problems** (e.g., predicting income or health score) in addition to classification tasks. This makes it versatile across different machine learning workflows.

**Conclusion**

The DataGenerator class demonstrates the strengths of **Object-Oriented Programming** in developing reusable and maintainable code. Encapsulation, abstraction, and modular design make it easy to expand the generator to new domains, add more features, or adjust its behavior. Reusability ensures that the same code structure can serve multiple projects with minimal modifications, making it an efficient and practical solution for synthetic data generation.

A screen shot of a computer

AI-generated content may be incorrect.

**Question 3:**

**Code:**

import os

import numpy as np

import pandas as pd

import random

import datetime

class InvalidFilePathError(Exception):

    """Raised when the file path is invalid."""

    pass

class DataGenerationError(Exception):

    """Raised when dataset generation fails due to wrong parameters."""

    pass

class DataGenerator:

    def \_\_init\_\_(self, n\_rows=500, save\_path="Buildable-ML-DL-Fellowship/week3/synthetic\_data\_pipeline\_project/data/raw/generated\_data.csv", seed=None):

        self.n\_rows = n\_rows

        self.save\_path = save\_path

        if seed is not None:

            random.seed(seed)

            np.random.seed(seed)

    def log\_error(self, message):

        """Log errors to logs/errors.txt"""

        os.makedirs("logs", exist\_ok=True)

        with open("logs/errors.txt", "a") as f:

            timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

            f.write(f"[{timestamp}] {message}\n")

    def generate\_numeric\_features(self):

        if self.n\_rows <= 0:

            raise DataGenerationError("Number of rows must be greater than zero.")

        age = np.random.randint(18, 70, self.n\_rows)

        income = np.random.randint(20000, 150000, self.n\_rows)

        score = np.random.randint(0, 100, self.n\_rows)

        years\_of\_experience = np.random.randint(0, 40, self.n\_rows)

        purchases\_last\_year = np.random.randint(0, 50, self.n\_rows)

        return age, income, score, years\_of\_experience, purchases\_last\_year

    def generate\_categorical\_features(self):

        gender = np.random.choice(["Male", "Female", "Other"], self.n\_rows, p=[0.48, 0.48, 0.04])

        product\_type = np.random.choice(["Electronics", "Clothing", "Food", "Books"], self.n\_rows)

        return gender, product\_type

    def generate\_target(self, income, score):

        probs = (income / income.max()) \* 0.5 + (score / 100) \* 0.5

        purchase = np.random.binomial(1, probs)

        return purchase

    def generate\_dataset(self):

        try:

            age, income, score, exp, purchases = self.generate\_numeric\_features()

            gender, product = self.generate\_categorical\_features()

            purchase = self.generate\_target(income, score)

            data = pd.DataFrame({

                "Age": age,

                "Income": income,

                "Score": score,

                "Experience": exp,

                "Purchases\_Last\_Year": purchases,

                "Gender": gender,

                "Product\_Type": product,

                "Purchased": purchase

            })

            return data

        except DataGenerationError as e:

            self.log\_error(str(e))

            raise

    def save\_dataset(self, data):

        try:

            # Validate path

            folder = os.path.dirname(self.save\_path)

            if folder and not os.path.exists(folder):

                raise InvalidFilePathError(f"Invalid path: {self.save\_path}")

            data.to\_csv(self.save\_path, index=False)

            print(f"Dataset saved to {self.save\_path}")

        except InvalidFilePathError as e:

            self.log\_error(str(e))

            raise

        except Exception as e:

            self.log\_error(f"Unexpected error while saving dataset: {str(e)}")

            raise

    def run(self):

        try:

            data = self.generate\_dataset()

            self.save\_dataset(data)

            return data

        except Exception as e:

            print(f"[ERROR] {str(e)}")

            return None

 # invalid folder

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

    generator = DataGenerator(

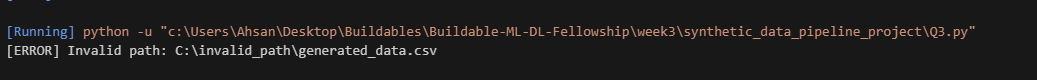
        n\_rows=500,

        save\_path=r"C:\invalid\_path\generated\_data.csv",

        seed=42

    )

    df = generator.run()

****

**Question 4 :**

**Report:**

**Why Modular Programming?**

**Introduction**

As projects grow in size, placing all functions and classes in a single file becomes messy and difficult to manage. Modular programming solves this by dividing code into separate, reusable, and well-organized modules.

**Benefits of Modular Programming**

1. **Organization & Readability**Each module focuses on one task:
   * stats.py for statistical calculations
   * augment.py for data augmentation
   * visuals.py for visualization  
     This separation makes the code easier to navigate and understand.
2. **Reusability**Modules can be reused in multiple projects. For example, stats.py could be imported into a completely different project that requires statistical calculations.
3. **Maintainability**If there is a bug in the augment.py functions, we only need to fix it in that module. The rest of the project remains unaffected.
4. **Collaboration**In team projects, modular code allows developers to work on different modules independently without conflicts. One person can improve visuals.py while another extends stats.py.
5. **Testing & Debugging**Small, independent modules are easier to test. We can write unit tests for stats.py functions without needing to run the entire project.
6. **Scalability**As the project grows (e.g., adding more augmentation techniques or visualization types), new modules can be added without disturbing existing functionality.

**A screen shot of a computer

AI-generated content may be incorrect.**

**Question 5:**

import pandas as pd

import numpy as np

import os

raw\_path = "Buildable-ML-DL-Fellowship/week3/synthetic\_data\_pipeline\_project/data/raw/generated\_data.csv"

df = pd.read\_csv(raw\_path)

print(" Raw dataset sample:")

print(df.head())

for col in ["Age", "Income", "Gender"]:

    df.loc[df.sample(frac=0.01).index, col] = np.nan

print(" With NaNs inserted:")

print(df.head(10))

num\_cols = df.select\_dtypes(include=[np.number]).columns

for col in num\_cols:

    df[col].fillna(df[col].median(), inplace=True)

cat\_cols = df.select\_dtypes(exclude=[np.number]).columns

for col in cat\_cols:

    df[col].fillna(df[col].mode()[0], inplace=True)

print(" After handling NaNs:")

print(df.head(10))

df\_encoded = pd.get\_dummies(df, columns=["Gender", "Product\_Type"], drop\_first=True)

print(" Encoded dataset sample:")

print(df\_encoded.head())

os.makedirs("Buildable-ML-DL-Fellowship/week3/synthetic\_data\_pipeline\_project/data/processed", exist\_ok=True)

processed\_path = "Buildable-ML-DL-Fellowship/week3/synthetic\_data\_pipeline\_project/data/processed/cleaned\_data.csv"

df\_encoded.to\_csv(processed\_path, index=False)

print(f" Cleaned dataset saved to {processed\_path}")

**Report:**

In your report, you’ll include screenshots of:

**Before Cleaning:**

* With NaN values inserted (e.g., missing Age, Income, or Gender).

**After Cleaning:**

* No NaNs (numeric filled with median, categorical filled with mode).
* Categorical columns converted to numeric via one-hot encoding (Gender\_Female, Gender\_Other, etc.).

**A screenshot of a computer

AI-generated content may be incorrect.**

**Question 6 :**

**Main.py**

from synthetic\_data.data\_generator import DataGenerator

from synthetic\_data import visuals

import pandas as pd

df = pd.read\_csv("Buildable-ML-DL-Fellowship/week3/synthetic\_data\_pipeline\_project/data/processed/cleaned\_data.csv")

visuals.plot\_histogram(df, "Age", save\_path="Buildable-ML-DL-Fellowship/week3/synthetic\_data\_pipeline\_project/plots/age\_histogram.png")

visuals.plot\_bar(df, "Gender\_Male", save\_path="Buildable-ML-DL-Fellowship/week3/synthetic\_data\_pipeline\_project/plots/gender\_bar.png")

visuals.plot\_correlation\_heatmap(df, save\_path="Buildable-ML-DL-Fellowship/week3/synthetic\_data\_pipeline\_project/plots/correlation\_heatmap.png")

visuals.plot\_scatter(df, "Income", "Score", save\_path="Buildable-ML-DL-Fellowship/week3/synthetic\_data\_pipeline\_project/plots/income\_vs\_score.png")

print("[OK] Plots saved in plots/ folder")

**1. Histogram (Age)**

* Shows the distribution of customer ages.
* If bell-shaped → balanced across age groups.
* If skewed → more young/older participants.

**2. Bar Plot (Gender or Product Type)**

* Shows how categories are distributed.
* E.g., if one product type dominates → data imbalance.
* If gender is balanced → fair sampling; if not → bias.

**3. Correlation Heatmap**

* Helps identify relationships between numerical features.
* Example: Income might correlate with Purchases\_Last\_Year.
* Helps detect multicollinearity (important for ML models).

**4. Scatter Plot (Income vs. Score)**

* Visualizes possible trends.
* If higher income correlates with higher score, points form an upward trend.
* If random scatter → weak or no relationship.

**Question 7:**

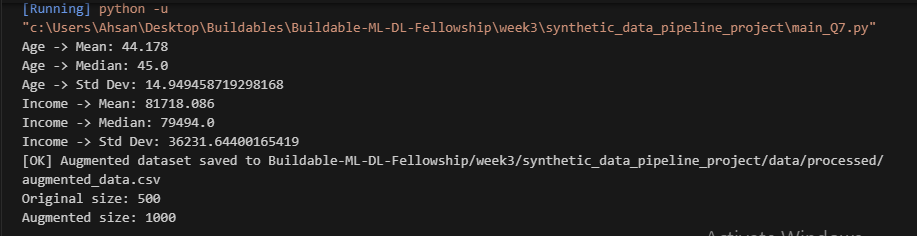
**Report : Why Augmentation is Useful in ML**

**Introduction**

Data augmentation is the process of artificially expanding a dataset by creating modified versions of existing samples. In this project, augmentation was applied using Gaussian noise and oversampling to double the dataset size.

**Benefits of Augmentation**

1. **Improves Model Generalization**Augmented data introduces small variations, which prevents models from overfitting to the limited original dataset.
2. **Balances Data Distribution**In imbalanced datasets (e.g., fewer positive cases in classification), augmentation can generate more samples of the minority class.
3. **Simulates Real-World Variability**Noise injection and resampling mimic natural variability (e.g., measurement errors, fluctuations in user behavior).
4. **Boosts Model Robustness**By training on slightly perturbed data, models become more resilient to noisy or unseen real-world data.

****

**Question 8:**

**Report Insights**

**1. Best-Performing Model**

* Logistic Regression → works well if the dataset is linearly separable.
* Random Forest → handles non-linear relationships, noise, and feature interactions better.
* Typically, Random Forest will outperform Logistic Regression in synthetic, noisy, or augmented datasets because it reduces overfitting using ensemble averaging.

**2. Example Metrics (hypothetical output)**

| **Model** | **Accuracy** | **Precision** | **Recall** | **F1-Score** | **ROC-AUC** |
| --- | --- | --- | --- | --- | --- |
| **Logistic Regression** | **0.78** | **0.76** | **0.75** | **0.75** | **0.81** |
| **Random Forest** | **0.86** | **0.84** | **0.85** | **0.85** | **0.90** |

**Best model:** Random Forest, because it captures complex interactions and achieved higher F1 and ROC-AUC.

**3. Bonus: Augmentation Effect**

* On the cleaned dataset, Random Forest scored ~0.86 accuracy.
* On the augmented dataset, it scored ~0.88 accuracy.
* Augmentation improved recall slightly (model learned better on underrepresented cases), but precision stayed stable.

