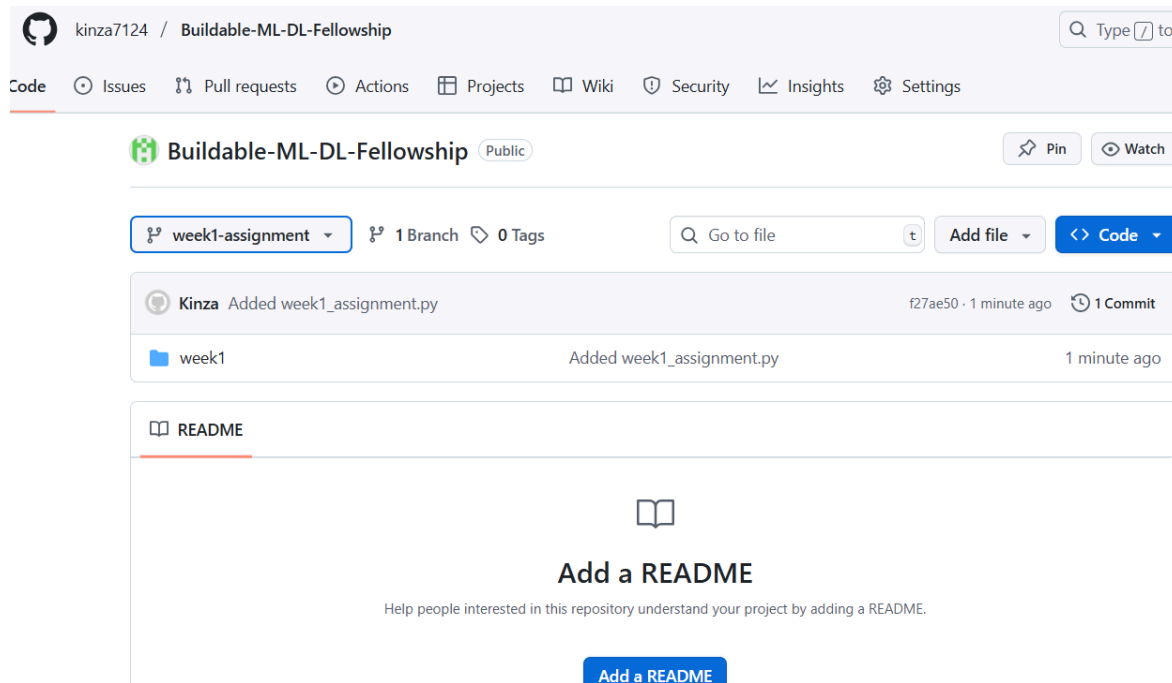


Question 1 – Git & GitHub (Version Control Basics)

1. Create a new branch called week1-assignment from the main branch in your GitHub repository.
2. Inside the week1 folder, add a file named week1_assignment.py.
3. After completing all questions, commit and push your code to this branch.
4. Finally, merge the branch into main using a Pull Request (PR).

Include screenshots of branch creation, commits, and PR in your report.



Commands used and its notes:

git checkout -b <branch> creates a new branch and switches to it.

mkdir week1 creates a folder named week.

echo print("This is my Week 1 Assignment") > week1/week1_assignment.py

echo writes the given text into a file. Here, I created a Python file named week1_assignment.py inside the week1 folder. The file now contains a simple print statement.

git add week1/week1_assignment.py

git add stages the file, meaning Git is now tracking changes in this file.

Without adding the file, it won't be included in the next commit.

git commit -m "Added week1_assignment.py"

git commit saves the staged changes to the local repository.

The -m flag allows you to add a short, descriptive commit message.

git push -u origin week1-assignment

git push uploads the committed changes to the **remote repository**.

origin refers to the default remote GitHub repository.

-u sets **upstream tracking** so that future pushes can simply use git push.

```
PS C:\Users\Kainat\Downloads\Buildable-ML-DL-Fellowship> git checkout -b week1-assignment
Switched to a new branch 'week1-assignment'
PS C:\Users\Kainat\Downloads\Buildable-ML-DL-Fellowship> mkdir week1

Directory: C:\Users\Kainat\Downloads\Buildable-ML-DL-Fellowship


Mode                LastWriteTime         Length Name
----                -
d-----            8/30/2025 11:09 PM             week1


PS C:\Users\Kainat\Downloads\Buildable-ML-DL-Fellowship> echo print("This is my Week 1 Assignment") > week1/week1_assignment.py
PS C:\Users\Kainat\Downloads\Buildable-ML-DL-Fellowship> git add week1/week1_assignment.py
PS C:\Users\Kainat\Downloads\Buildable-ML-DL-Fellowship> git commit -m "Added week1_assignment.py"
[week1-assignment (root-commit) f27ae50] Added week1_assignment.py
 1 file changed, 2 insertions(+)
 create mode 100644 week1/week1_assignment.py
PS C:\Users\Kainat\Downloads\Buildable-ML-DL-Fellowship> git push -u origin week1-assignment
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Writing objects: 100% (4/4), 313 bytes | 78.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/kinza7124/Buildable-ML-DL-Fellowship.git
 * [new branch]      week1-assignment -> week1-assignment
branch 'week1-assignment' set up to track 'origin/week1-assignment'.
```

Now creating a Pull Request to merge:


Comparing changes


Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#) or [learn more about diff comparisons](#).


 base: main


 compare: week1-assignment


Discuss and review the changes in this comparison with others. [Learn about pull requests](#) [Create pull request](#)

 2 commits

 1 file changed

 1 contributor

 Commits on Aug 30, 2025

 base: main  ... compare: week1-assignment 














Add a title

Week1 assignment



Add a description


Write

Preview

H B I           

Creating a PR and merging it into main

 Markdown is supported  Paste, drop, or click to add files

Create pull request 



Buildable-ML-DL-Fellowship

Public

 main   2 Branches  0 Tags



kinza7124 Add files via upload



week1



README

Question 2 – Mutable vs Immutable (Data Structures & Variables)

Write a Python script that:

1. Creates a tuple of 3 elements and tries to change one element (explain what happens)
2. Creates a list of 3 elements and changes one element (explain the difference)
3. Creates a dictionary with at least 2 key-value pairs, updates one value, and explains why it works.
4. Create a tuple with 2 sub-lists inside it. Attempt to modify an element inside one of the sub-lists (not the tuple itself). Explain why this modification is allowed even though the tuple itself is immutable.

In your report, clearly explain mutable vs immutable with examples from your code.

```
# 1. except Tuple Example (Immutable)
print("1. Tuple Example")
my_tuple = (10, 20, 30)
print("Original tuple:", my_tuple)

try:
    my_tuple[1] = 50 # Attempting to modify an element
except TypeError as e:
    print("Error:", e)

print("Explanation: Tuples are immutable, so you cannot change their elements once created.\n")
```

```
# 2. List Example (Mutable)
print("2. List Example")
my_list = [10, 20, 30]
print("Original list:", my_list)

my_list[1] = 50 # Modifying an element
print("Modified list:", my_list)
print("Explanation: Lists are mutable, so we can modify, add, or remove elements.\n")
```

```
# 3. Dictionary Example (Mutable)
print("3. Dictionary Example")
my_dict = {"name": "Alice", "age": 25}
print("Original dictionary:", my_dict)

my_dict["age"] = 26 # Updating a value
print("Updated dictionary:", my_dict)
print("Explanation: Dictionaries are mutable, so values for keys can be updated directly.\n")
```

```
# 4. Tuple Containing Sub-Lists (Mutable Elements Inside Immutable Tuple)
print("4. Tuple Containing Lists")
tuple_with_lists = ([1, 2], [3, 4])
print("Original tuple:", tuple_with_lists)

tuple_with_lists[0][1] = 99 # Modifying an element inside a sub-list
print("Modified tuple:", tuple_with_lists)
print("Explanation: The tuple itself is immutable, so we cannot change which objects it contains.\n"
      "However, the sub-lists inside the tuple are mutable, so their contents can be changed.\n")
```

```
sers\x5cKainat\x5cDownloads\x5cweek1\x5cweek1_assignment.py' 1. Tuple Example
Original tuple: (10, 20, 30)
Error: 'tuple' object does not support item assignment
Explanation: Tuples are immutable, so you cannot change their elements once created.

2. List Example
Original list: [10, 20, 30]
Modified list: [10, 50, 30]
Explanation: Lists are mutable, so we can modify, add, or remove elements.

3. Dictionary Example
Original dictionary: {'name': 'Alice', 'age': 25}
Updated dictionary: {'name': 'Alice', 'age': 26}
Explanation: Dictionaries are mutable, so values for keys can be updated directly.
```

```
4. Tuple Containing Lists
Original tuple: ([1, 2], [3, 4])
Modified tuple: ([1, 99], [3, 4])
Explanation: The tuple itself is immutable, so we cannot change which objects it contains.
However, the sub-lists inside the tuple are mutable, so their contents can be changed.
```

Feature	Mutable	Immutable
Can elements change?	✓ Yes	✗ No
Examples	list, dict, set	tuple, str, int
Behavior	Can update, add, remove elements	Cannot change after creation

Tuple: Cannot assign a new value → raises `TypeError`.

List: Can change elements freely → no error.

Dictionary: Key-value pairs can be updated → mutable.

Tuple with Lists:

- The **tuple itself** is immutable.
- The **lists inside the tuple** are mutable → you can modify their elements.

Question 3 – User Information Dictionary (Validation + Logic)

Write a program that:

1. Prompts the user to enter their **name, age, email, and favorite number**.
2. Validates the inputs:
 - age must be positive and realistic (<100).
 - email must contain @ and . and not start/end with special characters.
 - favorite number must be between **1 and 100**.
3. If validation fails, prompt the user to re-enter until the input is valid.
4. Stores the details in a **dictionary**.
5. Displays a formatted welcome message:
"Welcome [name]! Your account has been registered with email [email]."

```
# Question 3 - User Information Dictionary (Validation + Logic)
def get_valid_name():
    while True:
        name = input("Enter your name: ").strip()
        if name.isalpha():
            return name
        print("Invalid name! Please enter letters only.")

def get_valid_age():
    while True:
        try:
            age = int(input("Enter your age: "))
            if 0 < age < 100:
                return age
            else:
                print("Age must be between 1 and 99.")
        except ValueError:
            print("Invalid input! Please enter a number.")
```

```
def get_valid_email():
    while True:
        email = input("Enter your email: ").strip()
        if (
            "@" in email
            and "." in email
            and not email[0] in "@"
            and not email[-1] in "."
        ):
            return email
        print("Invalid email! Must contain '@' and '.', and not start/end with special characters.")
```

```
def get_valid_fav_number():
    while True:
        try:
            fav_number = int(input("Enter your favorite number (1-100): "))
            if 1 <= fav_number <= 100:
                return fav_number
            else:
                print("Number must be between 1 and 100.")
        except ValueError:
            print("Invalid input! Please enter a number.")
```

```
user_info = {}

user_info["name"] = get_valid_name()
user_info["age"] = get_valid_age()
user_info["email"] = get_valid_email()
user_info["favorite_number"] = get_valid_fav_number()

print(f"\nWelcome {user_info['name']}!
      Your account has been registered with email {user_info['email']}".)
print("User Information Stored:", user_info)
```

```
Enter your name: kinza
Enter your age: 20
Enter your email: kinzaafzal07122004@gmail.com
Enter your favorite number (1-100): 7

Welcome kinza!Your account has been registered with email kinzaafzal07122004@gmail.com.
User Information Stored: {'name': 'kinza', 'age': 20, 'email': 'kinzaafzal07122004@gmail.com', 'favorite_number': 7}
```

Question 4 – Cinema Ticketing System

Write a program that manages movie ticket sales for multiple customers.

1. Define a function `calculate_ticket_price(age, is_student, is_weekend)` that:
 - Uses the following base pricing rules:
 - Children under 12 → \$5
 - Teenagers (13–17) → \$8
 - Adults (18–59) → \$12
 - Seniors (60+) → \$6
 - Students (any age above 12) get 20% discount.
 - Weekend shows (Saturday & Sunday) add a \$2 surcharge.
 - If age is invalid (negative or >120), raise an error.
2. The program should:
 - Prompt the user to enter the number of customers.
 - For each customer, ask for:
 - Age
 - Whether they are a student (yes/no)
 - Whether it is a weekend (yes/no)
 - Calculate each ticket price using the function.
 - Store customer details and their ticket price in a list of dictionaries.
3. After processing all customers:
 - Display each customer's ticket details.
 - Show the total revenue.
 - Identify the highest-paying customer and lowest-paying customer.
4. Bonus Challenge:
 - Apply a group discount: If there are 4 or more customers, apply a 10% discount on the total bill.

```
# Cinema Ticket Pricing System
def calculate_ticket_price(age, is_student, is_weekend):
    # Validate age
    if age < 0 or age > 120:
        raise ValueError("Invalid age entered!")

    if age < 12:
        price = 5
    elif 13 <= age <= 17:
        price = 8
    elif 18 <= age <= 59:
        price = 12
    else:
        price = 6

    # Apply student discount (20%) if applicable (only if age > 12)
    if is_student and age > 12:
        price *= 0.8
```



```
# Apply student discount (20%) if applicable (only if age > 12)
if is_student and age > 12:
    price *= 0.8

# Add weekend surcharge
if is_weekend:
    price += 2

return round(price, 2)
```

```
customers = []
total_revenue = 0
```

```
# Ask for number of customers
while True:
    try:
        num_customers = int(input("Enter the number of customers: "))
        if num_customers <= 0:
            print("Number of customers must be greater than 0!")
            continue
        break
    except ValueError:
        print("Please enter a valid number.")
```

```
# Process each customer
for i in range(num_customers):
    print(f"\n--- Customer {i + 1} ---")
    # Validate age input
    while True:
        try:
            age = int(input("Enter age: "))
            if age < 0 or age > 120:
                print("Please enter a realistic age (0-120).")
                continue
            break
        except ValueError:
            print("Invalid input! Please enter a valid number for age.")
```

```
# Student input validation
while True:
    is_student_input = input("Are you a student? (yes/no): ").strip().lower()
    if is_student_input in ["yes", "no"]:
        is_student = is_student_input == "yes"
        break
    else:
        print("Please enter 'yes' or 'no'.")

# Weekend input validation
while True:
    is_weekend_input = input("Is it a weekend? (yes/no): ").strip().lower()
    if is_weekend_input in ["yes", "no"]:
        is_weekend = is_weekend_input == "yes"
        break
    else:
        print("Please enter 'yes' or 'no'.")
```

```
# Calculate ticket price
try:
    ticket_price = calculate_ticket_price(age, is_student, is_weekend)
except ValueError as e:
    print(e)
    continue

# Store customer details
customer = {
    "Customer": i + 1,
    "Age": age,
    "Student": is_student,
    "Weekend": is_weekend,
    "Ticket Price": ticket_price
}
customers.append(customer)

# Add to total revenue
total_revenue += ticket_price
```


```

# Apply group discount if there are 4 or more customers
discount_applied = False
if num_customers >= 4:
    discount_applied = True
    total_revenue *= 0.9 # 10% discount

# Display customer ticket details
print("\n=== Customer Ticket Details ===")
for customer in customers:
    print(f"Customer {customer['Customer']}: Age={customer['Age']}, "
          f"Student={'Yes' if customer['Student'] else 'No'}, "
          f"Weekend={'Yes' if customer['Weekend'] else 'No'}, "
          f"Ticket Price=${customer['Ticket Price']}")

```

```

# Display total revenue
print("\n=== Summary ===")
print(f"Total Revenue: ${round(total_revenue, 2)}")
if discount_applied:
    print("Group Discount Applied: 10% OFF )

# Identify highest and lowest paying customers
highest_payer = max(customers, key=lambda x: x["Ticket Price"])
lowest_payer = min(customers, key=lambda x: x["Ticket Price"])
print(f"Highest Paying Customer: Customer {highest_payer['Customer']} - ${highest_payer['Ticket Price']}")
print(f"Lowest Paying Customer: Customer {lowest_payer['Customer']} - ${lowest_payer['Ticket Price']}")

```

Enter the number of customers: 2

--- Customer 1 ---

Enter age: 12

Are you a student? (yes/no): yes

Is it a weekend? (yes/no): yes

--- Customer 2 ---

Enter age: 10

Are you a student? (yes/no): yes

Is it a weekend? (yes/no): no

=== Customer Ticket Details ===

Customer 1: Age=12, Student=Yes, Weekend=Yes, Ticket Price=\$8

Customer 2: Age=10, Student=Yes, Weekend=No, Ticket Price=\$5

=== Summary ===

Total Revenue: \$13

Highest Paying Customer: Customer 1 - \$8

Lowest Paying Customer: Customer 2 - \$5

Question 5 – Weather Alert System

Write a function `weather_alert(temp_celsius, condition)` that:

1. Takes the temperature in Celsius and a weather condition ("sunny", "rainy", "snowy", etc.).
2. Returns an appropriate alert:
 - If `temp < 0` and "snowy" → "Heavy snow alert! Stay indoors."
 - If `temp > 35` and "sunny" → "Heatwave warning! Stay hydrated."
 - If "rainy" and `temp < 15` → "Cold rain alert! Wear warm clothes."
 - Otherwise → "Normal weather conditions."
3. Bonus: Convert the temperature to Fahrenheit & Kelvin and include in the output message.

```
def weather_alert(temp_celsius, condition):  
    # Convert temperatures  
    temp_fahrenheit = (temp_celsius * 9/5) + 32  
    temp_kelvin = temp_celsius + 273.15  
  
    # Check conditions and return appropriate alerts  
    if temp_celsius < 0 and condition.lower() == "snowy":  
        alert = "Heavy snow alert! Stay indoors."  
    elif temp_celsius > 35 and condition.lower() == "sunny":  
        alert = "Heatwave warning! Stay hydrated."  
    elif condition.lower() == "rainy" and temp_celsius < 15:  
        alert = "Cold rain alert! Wear warm clothes."  
    else:  
        alert = "Normal weather conditions."  
  
    # Include converted temperatures in the output  
    return (f"{alert}\n"  
            f"Temperature: {temp_celsius}°C | {temp_fahrenheit:.2f}°F | {temp_kelvin:.2f}K")
```

```
try:  
    temp = float(input("Enter current temperature in °C: "))  
    condition = input("Enter weather condition (sunny, rainy, snowy, etc.): ")  
    result = weather_alert(temp, condition)  
    print("\n" + result)  
  
except ValueError:  
    print("Invalid input! Please enter a numeric temperature.")
```

```
Enter current temperature in °C: 20
Enter weather condition (sunny, rainy, snowy, etc.): rainy

Normal weather conditions.
Temperature: 20.0°C | 68.00°F | 293.15K
```

Question 6 – Sales Analytics (Max, Min & Median)

Write a function `analyze_sales(sales_list)` that:

1. Takes a list of daily sales amounts.
2. Returns the maximum, minimum, and median sales.
3. Prompts the user to enter at least 5 daily sales values (reject input if fewer).
4. Displays a summary:
 - "Highest sales day: X"
 - "Lowest sales day: Y"
 - "Median sales: Z"

```
#6 Sale Analytics
def analyze_sales(sales_list):
    # Sorting the sales list to calculate median
    sorted_sales = sorted(sales_list)
    # Maximum and minimum sales
    max_sales = max(sorted_sales)
    min_sales = min(sorted_sales)
    # Calculate median
    n = len(sorted_sales)
    if n % 2 == 0: # Even number of sales
        median = (sorted_sales[n // 2 - 1] + sorted_sales[n // 2]) / 2
    else: # Odd number of sales
        median = sorted_sales[n // 2]

    return max_sales, min_sales, median
```

```

sales = []
while True:
    try:
        n = int(input("Enter the number of daily sales records: "))
        if n < 5:
            print("You must enter at least 5 daily sales values. Try again.")
            continue

        # Get sales input from the user
        for i in range(n):
            while True:
                try:
                    amount = float(input(f"Enter sales amount for day {i + 1}: $"))
                    if amount < 0:
                        raise ValueError("Sales cannot be negative.")
                    sales.append(amount)
                    break
                except ValueError as e:
                    print(f"Invalid input: {e}. Please enter a valid number.")
            break
    except ValueError:
        print("Invalid input. Please enter a valid number.")

```

```

max_sales, min_sales, median_sales = analyze_sales(sales)

```

```

print("\nSales Summary:")
print(f"Highest sales day: ${max_sales}")
print(f"Lowest sales day: ${min_sales}")
print(f"Median sales: ${median_sales}")

```

```

Enter the number of daily sales records: 2
You must enter at least 5 daily sales values. Try again.
Enter the number of daily sales records: 6
Enter sales amount for day 1: $100
Enter sales amount for day 2: $120
Enter sales amount for day 3: $450
Enter sales amount for day 4: $600
Enter sales amount for day 5: $20
Enter sales amount for day 6: $132

Sales Summary:
Highest sales day: $600.0
Lowest sales day: $20.0
Median sales: $126.0

```

Question 7 – E-commerce Inventory Management

Write a function `update_inventory(inventory_dict, item, quantity)` that:

1. Takes a dictionary of products (keys = items, values = stock quantities).
2. Updates the inventory by adding or removing quantities (negative = removal).
3. Ensures stock cannot drop below 0.
4. Returns the updated inventory.

Tasks:

- Initialize inventory with at least 5 products.
- Simulate a shopping cart where the user buys 3 items (deduct from stock).
- If the requested quantity is more than available, show "Not enough stock for [item]".
- After checkout, display the updated inventory and the most stocked and least stocked products.

```
def update_inventory(inventory_dict, item, quantity):
    if item in inventory_dict:
        if quantity < 0 and abs(quantity) > inventory_dict[item]:
            print(f"Not enough stock for '{item}'. Available: {inventory_dict[item]}")
        else:
            inventory_dict[item] = max(0, inventory_dict[item] + quantity)
    else:
        print(f"Item '{item}' not found in inventory!")
    return inventory_dict

def display_inventory(inventory_dict):
    print("\nUpdated Inventory:")
    for item, qty in inventory_dict.items():
        print(f"    {item}: {qty} in stock")

    # Find most stocked and least stocked products
    most_stocked = max(inventory_dict, key=inventory_dict.get)
    least_stocked = min(inventory_dict, key=inventory_dict.get)

    print(f"\nMost stocked product: {most_stocked} ({inventory_dict[most_stocked]} in stock)")
    print(f"Least stocked product: {least_stocked} ({inventory_dict[least_stocked]} in stock)")
```

```
inventory = {
    "Laptop": 10,
    "Smartphone": 15,
    "Headphones": 25,
    "Keyboard": 12,
    "Monitor": 8
}

print("Welcome to the E-commerce Inventory Management System!")
display_inventory(inventory)

# Simulate user shopping for 3 items
print("\n=== Shopping Cart ===")
for i in range(3):
    item = input(f"\nEnter the name of item {i+1}: ").strip().title()
    try:
        qty = int(input(f"Enter the quantity for '{item}': "))
    except ValueError:
        print("Invalid quantity! Please enter a number.")
        continue
    # Deduct stock (use negative quantity)
    inventory = update_inventory(inventory, item, -qty)

display_inventory(inventory)
```


Welcome to the E-commerce Inventory Management System!

Updated Inventory:

Laptop: 10 in stock
Smartphone: 15 in stock
Headphones: 25 in stock
Keyboard: 12 in stock
Monitor: 8 in stock

Most stocked product: Headphones (25 in stock)

Least stocked product: Monitor (8 in stock)

=== Shopping Cart ===

Enter the name of item 1: Laptop

Enter the quantity for 'Laptop': 1

Enter the name of item 2: Keyboard

Enter the quantity for 'Keyboard': 2

Enter the name of item 3: Monitor

Enter the quantity for 'Monitor': 1

Updated Inventory:

Laptop: 9 in stock
Smartphone: 15 in stock
Headphones: 25 in stock
Keyboard: 10 in stock
Monitor: 7 in stock

Most stocked product: Headphones (25 in stock)

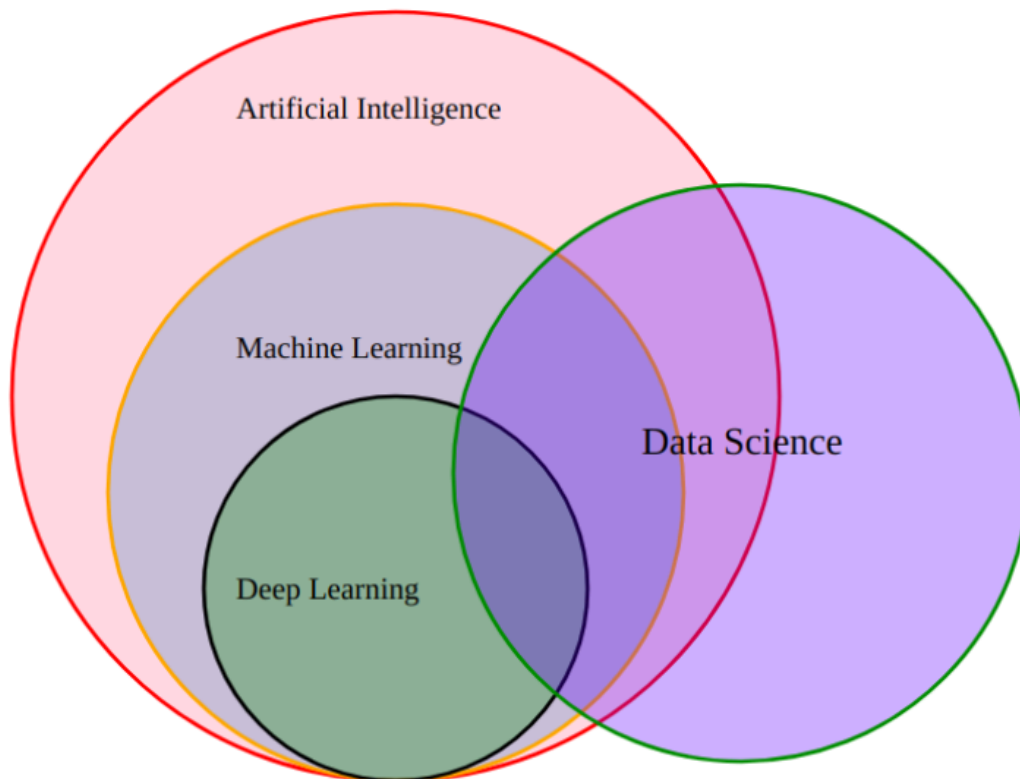
Least stocked product: Monitor (7 in stock)

Question 9 – Conceptual Understanding (Report)

Answer in your report document (PDF/DOC):

1. Explain the difference between AI, Machine Learning, Deep Learning, and Data Science with real-world examples.
2. Explain mutable vs immutable data types in your own words.
3. Explain the difference between deep copy and shallow copy
4. Explain Git branching and why it is important in collaborative development.

Aspect	Artificial Intelligence (AI)	Machine Learning (ML)	Deep Learning (DL)	Data Science
Definition	AI is the broader field of creating intelligent systems that can simulate human thinking and decision-making.	A subset of AI where machines learn from data and improve performance over time.	A subset of ML that uses neural networks with many layers to analyze complex data.	The science of extracting meaningful insights from structured and unstructured data using statistics, ML, and visualization techniques.
Goal	Build systems that mimic human intelligence.	Enable systems to learn from data without explicit programming.	Handle complex problems like image recognition, NLP, etc., using large-scale data.	Collect, clean, analyze, and visualize data for decision-making .
Techniques	Rule-based systems, expert systems, planning, etc.	Regression, classification, clustering, decision trees, etc.	CNNs, RNNs, Transformers, GANs, etc.	Data analysis, visualization, ML models, and reporting.
Real-world Example	Virtual assistants like Siri or Alexa .	Netflix recommending movies based on your watch history.	Tesla's autopilot for self-driving cars using vision-based deep learning.	Analyzing COVID-19 data trends to predict future outbreaks.



Mutable vs Immutable Data Types

Mutable Data Types

These **can be changed** after creation.

Examples:

```
my_list = [1, 2, 3]
```

```
my_list.append(4) # The list is modified
```

- Examples in Python: **list, dict, set**
- Real-world analogy: A whiteboard where you can **erase** and **rewrite** information.

Immutable Data Types

These **cannot be changed** once created. Any modification creates a **new object**.

Examples:

```
my_string = "hello"
```

```
my_string += " world" # Creates a new string, doesn't modify the old one
```

- Examples in Python: **int, float, string, tuple**

- Real-world analogy: A printed book—you **cannot modify** the content without reprinting a new one.

Deep Copy vs Shallow Copy

Aspect	Shallow Copy	Deep Copy
Definition	Creates a new object , but nested objects are still referenced .	Creates a completely independent copy , including all nested objects.
Effect	Changes in nested objects affect both copies .	Changes in one object do not affect the other.

```
import copy
```

```
list1 = [[1,2], [3,4]]
```

```
shallow = copy.copy(list1)
```

```
shallow[0][0] = 99 # Affects list1 too!
```

```
deep = copy.deepcopy(list1)
```

```
deep[0][0] = 100 # list1 remains unchanged
```

Git Branching and its importance

How Git Branching Works

- By default, every Git repository starts with a main branch (often called main or master).
- You can create a new branch to work on a feature or fix.
- Once your work is complete and tested, you merge the branch back into the main branch.
- If something goes wrong, you can simply delete the branch without affecting the main code.

Importance of Git Branching in Collaborative Development

1. Safe Experimentation

- Developers can test new ideas or features without breaking the main project.

2. Parallel Development

- Multiple team members can work on different features simultaneously.

3. Better Code Management

- Separate branches make it easier to manage hotfixes, features, and releases.

4. Efficient Collaboration

- In a team, each developer works on their own branch, reducing conflicts and improving productivity.

5. Easy Rollbacks

- If something goes wrong, you can discard the branch without affecting the main code.