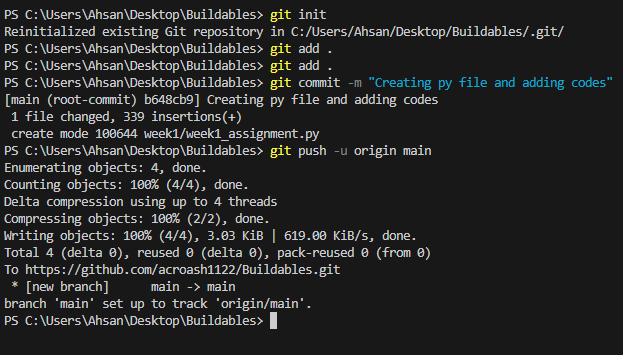
**Question 1:**

****

* I just create a repo from github and clone in local.
* After that I add documents and following code in it . then proceed with add, commit and push to github
* Named the branch named as **week1-assignment.**
* Now Merge that branch into the main branch.

**Question** 2:

a)

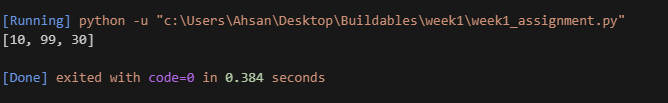
A screen shot of a computer

AI-generated content may be incorrect.

**Explanation:**

* Tuples are **immutable** → once created, you cannot change, add, or remove elements.
* That’s why my\_tuple[1] = 99 raises a **TypeError**.

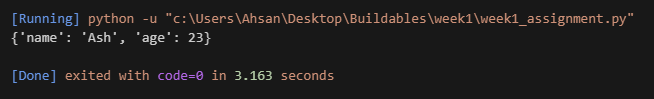
b)



**Explanation:**

* Lists are **mutable**, so we can directly modify, add, or remove elements.
* Unlike tuples, this change is allowed, and the list now reflects the update.

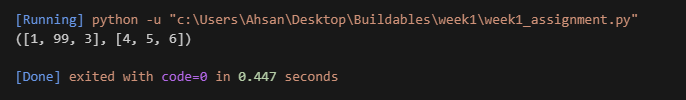
c)



**Explanation:**

* Dictionaries are **mutable** → values can be changed without creating a new dictionary.
* When we update "age", Python just changes the stored value for that key.

d)



**Explanation:**

* The **tuple itself is immutable** (you cannot replace its elements, e.g., cannot assign a new list).
* But the **objects stored inside** (the lists) are **mutable**.
* That’s why we can still modify the contents of the list inside the tuple.

e)

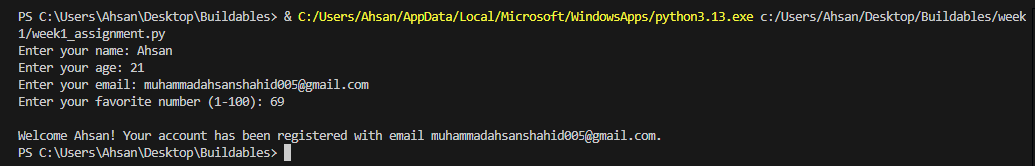
**Immutable objects:** Cannot be changed after creation.

* Examples: tuple, str, int, float.
* If you try to modify, Python raises an error.

**Mutable objects:** Can be modified in place.

* Examples: list, dict, set.
* You can change contents without recreating the object.

**Question 3**:



**Explanation:**

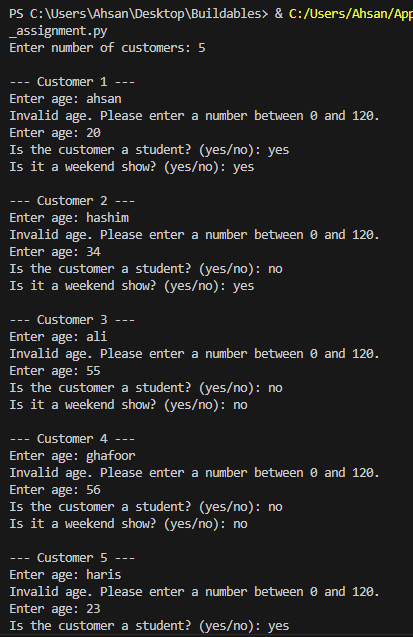
Here is step by step explanation.

* get\_valid\_name() -- makes sure name is not empty.
* get\_valid\_age() -- ensures age is an integer between 1 and 99.
* get\_valid\_email() -- checks for @, ., and ensures not starting/ending with special characters.
* get\_valid\_fav\_number() -- ensures number is between 1 and 100.

All valid values are stored in a **dictionary user\_info**.

Finally, a **formatted welcome message** is displayed.

**Question 4**:



A screen shot of a computer

AI-generated content may be incorrect.

**Explanation**

**Step 1: Break the Problem into Subtasks**

The problem is big, so first we **divide** it into manageable parts:

1. **Price Calculation Logic**
   * Depends on **age, student status, weekend status**.
   * This belongs inside a **function** (calculate\_ticket\_price).
2. **Input & Validation**
   * Ask the user for **age, student, weekend**.
   * Validate each input (age must be valid, yes/no answers should be correct).
3. **Store Customer Data**
   * Each customer’s details (age, student, weekend, ticket price) stored in a **dictionary**.
   * Keep all customers in a **list of dictionaries**.
4. **Reporting & Revenue**
   * Print each customer’s ticket details.
   * Calculate **total revenue**.
   * Find **highest-paying** and **lowest-paying** customers.
   * Apply **group discount** if ≥4 customers.

**Step 2: Ticket Pricing Rules (Core Function)**

This is the **heart of the system**.  
We create calculate\_ticket\_price(age, is\_student, is\_weekend).

Rules:

* **Children (<12)** → $5
* **Teenagers (13–17)** → $8
* **Adults (18–59)** → $12
* **Seniors (60+)** → $6
* **Students (>12)** → 20% discount
* **Weekend shows** → +$2 surcharge

Why separate this into a function?

* Reusable, testable, and makes main code cleaner.

**Step 3: Collecting Input**

* For each customer, we ask:
  1. Age (must be 0–120)
  2. Student? (yes/no)
  3. Weekend show? (yes/no)

Instead of repeating validation logic everywhere, we use while True loops to keep asking until input is valid.

**Step 4: Storing Data**

We use a **list of dictionaries** like this:

[

{"id": 1, "age": 10, "student": False, "weekend": True, "ticket\_price": 7},

{"id": 2, "age": 20, "student": True, "weekend": False, "ticket\_price": 9.6},

{"id": 3, "age": 65, "student": False, "weekend": False, "ticket\_price": 6}

]

Why a dictionary?

* Easy to label and access values (e.g., customer['ticket\_price']).
* Easier to print readable results.

**Step 5: Reporting & Revenue**

At the end:

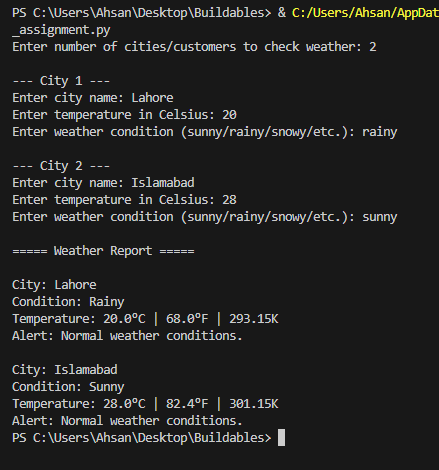
1. Print **each customer’s details**.
2. Use sum (or keep running total) for **revenue**.
3. Use max and min functions on the list to find **highest** and **lowest-paying customers**.
4. If there are 4 or more customers → apply **10% discount** on the total bill.

Why? This simulates **real business logic** (group discount = marketing strategy).

**Step 6: Example Run (to see it in action)**

The example run is in the screenshot.

**Question 5**:



**Weather Alert System**

**Step 1: Understand the Requirements**

* Function name: weather\_alert(temp\_celsius, condition)
* Inputs:
  + Temperature in Celsius
  + Condition string ("sunny", "rainy", "snowy", etc.)
* Outputs:
  + Weather alert message
  + Bonus: also include **Fahrenheit** and **Kelvin** equivalents

**Step 2: Formula for Temperature Conversion**

* **C → F**: F = (C \* 9/5) + 32
* **C → K**: K = C + 273.15

**Step 3: Define Rules in If-Else**

1. If **temp < 0 and condition == "snowy"** → Heavy snow alert
2. If **temp > 35 and condition == "sunny"** → Heatwave warning
3. If **condition == "rainy" and temp < 15** → Cold rain alert
4. Otherwise → Normal weather

**Step 4: Write the Function**

def weather\_alert(temp\_celsius, condition):

# Convert temperature

temp\_f = (temp\_celsius \* 9/5) + 32

temp\_k = temp\_celsius + 273.15

# Decide alert message

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."

# Final message with conversions

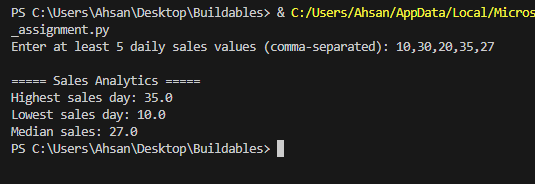
return (f"{alert}\n"

f"Temperature: {temp\_celsius}°C | {temp\_f:.1f}°F | {temp\_k:.2f}K")

**Step 5: Example Runs**

Mentioned in screenshots.

**Question 6**:



**Sales Analytics (Max, Min & Median)**

**Step 1: Requirements**

* Function analyze\_sales(sales\_list) → returns:
  + Maximum sales
  + Minimum sales
  + Median sales
* Program should:
  + Ask user to enter daily sales values.
  + Reject input if fewer than **5 values**.
  + Print summary in required format.

**Step 2: What is Median?**

* Median = Middle value in a sorted list.
* If **odd count** → pick middle element.
* If **even count** → average of two middle elements.

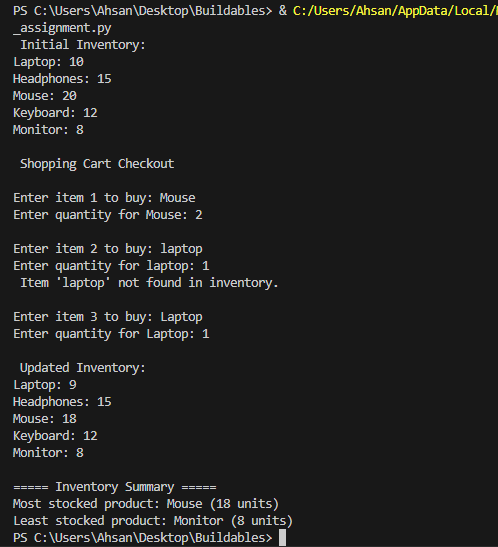
Example:

* [10, 20, 30] → Median = 20
* [10, 20, 30, 40] → Median = (20+30)/2 = 25

**Step 4: Example Run:**

Mentioned in screenshots.

**Question 7**:



**Explanation:**

1. **Data Model & Invariant**

* **Inventory structure:** dict[str, int] → e.g. {"Laptop": 10, "Mouse": 20, ...}
* **Invariant:** For every product p, inventory[p] >= 0 (stock can never go negative).

Why a dictionary?

* O(1) lookup/update by item name
* Human-readable keys (product names)

1. **Core Function: update\_inventory(inventory\_dict, item, quantity)**

**Contract**

* **Inputs:**
  + inventory\_dict: dictionary of current stocks
  + item: product name (string)
  + quantity: integer; **positive = add stock**, **negative = remove stock**
* **Behavior:**
  + If item not found → print error, do nothing
  + Compute new\_qty = current\_qty + quantity
  + If new\_qty < 0 → **reject** update and print:  
    Not enough stock for [item] (keeps inventory unchanged)
  + Else → set inventory[item] = new\_qty
* **Returns:** updated inventory\_dict

**Why this design?**

* Single function encapsulates **all stock mutation rules** and protects the invariant.
* Makes the main flow simple and readable.

1. **End-to-End Flow (Main Program)**
2. **Initialize** inventory with ≥ 5 products (e.g., Laptop, Headphones, Mouse, Keyboard, Monitor).
3. **Display** initial inventory.
4. **Simulate a cart of 3 purchases**: repeat 3 times  
   3.1 Ask user for item  
   3.2 Ask user for qty (must be a positive integer)  
   3.3 Call update\_inventory(inventory, item, -qty)  
   3.4 If not enough stock → show "Not enough stock for [item]" and **skip** the deduction
5. **Display** updated inventory.
6. **Analytics:**
   * **Most stocked**: max(inventory.items(), key=lambda kv: kv[1])
   * **Least stocked**: min(inventory.items(), key=lambda kv: kv[1])
7. **Print** both analytics.

**Question 9**

**1. Difference between AI, Machine Learning, Deep Learning, and Data Science**

Artificial Intelligence (AI): AI is the broad field of creating systems that can simulate human intelligence. Example: Chatbots like Siri or Alexa that understand and respond to user queries.  
  
Machine Learning (ML): ML is a subset of AI that uses data and algorithms to learn and improve automatically. Example: Email spam filters that learn to classify spam vs. non-spam emails.  
  
Deep Learning (DL): DL is a subset of ML that uses neural networks with many layers to handle complex tasks. Example: Self-driving cars that detect pedestrians and traffic signs using image recognition.  
  
Data Science: Data Science involves analyzing, visualizing, and interpreting data to gain insights and make decisions. Example: Netflix recommending movies based on viewing history.

**2. Mutable vs Immutable Data Types**

Mutable Data Types: These can be changed after creation. Example: Lists and Dictionaries in Python. You can add, remove, or modify elements.  
  
Immutable Data Types: These cannot be changed after creation. Example: Strings and Tuples in Python. If you try to modify them, a new object is created.

**3. Deep Copy vs Shallow Copy**

Shallow Copy: Creates a new object but references the same elements inside. Changes in nested objects affect both copies.  
  
Deep Copy: Creates a completely independent copy of the object, including nested elements. Changes in one do not affect the other.

**4. Git Branching in Collaborative Development**

Git branching allows developers to create separate versions of a project to work on new features, bug fixes, or experiments without affecting the main codebase. It is crucial for collaborative development because multiple developers can work on different branches simultaneously.  
  
Example: A 'main' branch contains production code, while a 'feature-login' branch is used to develop a new login feature. Once completed and tested, the branch can be merged back into the main branch.