✅ Important Instructions for NumPy-Based Assessments

🔍 1. Follow Function Names Strictly

- Use **exact function names** as mentioned in the problem statement.
- Python is **case-sensitive** —
  Example:
    - ✔️ create_stock_array is correct.
    - ❌ Create_Stock_Array, CreateStockArray, or createStockArray will cause **test failure**.

🔍 2. Be Careful with Return Types

If the problem asks for a **NumPy array** — you must return a NumPy array (np.ndarray), **not a list**.
Example:

python

```
return np.array([...])  # Correct
return [...]            # Incorrect
```

-
- If asked for a **tuple**, **return it as a tuple**, not as a list or single values.
- If the problem says return a **boolean** — return True or False, NOT strings "True" or "False".

🔍 3. Variable Names Matter (Case-Sensitive)

Respect case-sensitivity in variable names:

```python
order_array != Order_array != order_Array
```

- If the problem says "fuel_array" — use this name for consistency and clarity.

🔍 4. Proper Reading of Implementation Flow

- Many students **ignore the "Implementation Flow" section** — this leads to incomplete or incorrect logic.
- **You must follow every bullet point in Implementation Flow** — especially for:
    - Validation ranges (if array is empty return False)
    - Required calculations (round to 2 decimals using np.round())
    - Specific functions (np.sum, np.mean, np.max)
- These flows are there to guide you — read them fully.

🔍 5. Use NumPy Functions — Not Loops unless specified

- Always use NumPy built-in functions (np.sum, np.mean, np.where) as much as possible.

Do NOT use:

```python
for i in array:  # Unless the problem explicitly asks
```

- Reason: **NumPy is meant for vectorized, fast computation — not Python loops**.

🔍 6. Data Type Casting Mistakes

When creating arrays:

```python
np.array(order_list, dtype=np.float64)  # Use dtype when needed
```

- If the problem mentions float64 or int64, ensure this is followed.

🔍 7. Don’t Hardcode Values

Example mistake:

```python
mean = 100  # Incorrect unless problem says so
```

Always calculate using:

```python
mean = np.mean(order_array)  # Correct
```

🔍 8. Rounding Mistakes

If asked to round to 2 or 4 decimals:

```python
np.round(np.sum(arr), 2)  # Total rounded to 2 decimal
np.round(np.mean(arr), 4) # Mean rounded to 4 decimal
```

- Missing rounding leads to **test case failure**.

🔍 9. Condition-based Labeling — Use np.where / list comprehensions

Example Correct Approach:

python

```
np.where(arr >= 200, 'High', 'Normal')
```

- Wrong: Using manual loops or ignoring array-wise operations.

🔍 10. Formatting Mistakes

If problem says format like "XX.XX USD":

python

```
[f"${val:.2f}" for val in arr]
```

- Never skip such output expectations — otherwise output mismatch errors in tests.

🔍 11. Validation Logic is Mandatory

Example:
If validation requires checking negative numbers:

python

```
if np.any(arr < 0): return False
```

- Never skip input validation — most fail because they assume "input will always be correct".

🔍 12. Always Return the Final Output — Don't Print Unless Asked

- The problem says **"return the result"** — do not print().
- Printed outputs cannot be tested by auto-graders — only **returned outputs are checked**.

⚠️ Most Common Mistakes by Previous Batches:

Mistake	What Happens
Wrong function name	Pytest / checker fails — 0 marks
Returns list instead of NumPy array	Type error in checker
Missing validation checks	Test cases for invalid data fail
Manual loop instead of NumPy vectorization	Slower & logically incorrect
Rounding errors	Floating point mismatches
Ignored formatting (e.g. "$100.00")	Output mismatch, 0 marks
Prints instead of returns	Checker gets None — 0 marks

🎯 Final Pro Tips:

1. 📖 **Read the Problem Statement twice.**
2. ✅ Implement **only what is asked — no more, no less**.
3. 🏷️ Return **in exact format** — NumPy array, tuple, boolean, string.
4. ⚡ **Use NumPy functions**, avoid Python loops unless clearly needed.
5. ✔️ Validate all inputs if mentioned.
6. 🔍 Use **np.round(), np.where()**, and formatting strings correctly.
7. ❌ **Do NOT change function names or parameter names**.

📝 Example: Wrong vs Right

❌ **Wrong:**

```
def ComputeSummary(arr):  # Wrong function name
    return sum(arr)  # Python list sum — not NumPy
```

✔️ **Right:**

```
def compute_order_summary(self, order_array: np.ndarray) -> tuple:
    total = np.round(np.sum(order_array), 2)
    ...
```

⚠️ **AttributeError**

Why this happens:

- **You called a wrong method or property on a NumPy array or variable.**

Example of mistake:

```
arr = np.array([1,2,3])

arr.mean  # Missing ()
```

Error:

```
AttributeError: 'numpy.ndarray' object has no attribute 'mean'
```

Fix:
✅ **Methods must be called with brackets:**

python

```
arr.mean()  # Correct
```

✅ **Be careful with:**

- `.shape`, `.size` — **no brackets**
- `.mean()`, `.sum()` — **brackets needed**

---

4. ❌ Test Failed (Pytest or Manual Test Failure)

Why this happens:

- **Wrong return type (list instead of array or tuple instead of int)**
- **Function name spelling mistake**
- **Wrong format (e.g., missing rounding)**
- **Validation check skipped**
- **Extra prints or no return**

What to Check:
✅ **Function name matches problem exactly**
✅ **Return type is as expected (not print!)**
✅ **Rounded or formatted properly**
✅ **Did you handle empty arrays?**
✅ **If tuple expected — return a tuple**
✅ **Do not print, only return**

---

5. 🖨️ Print Statement Does Not Work in Test Cases

Why this happens:

- **Pytest or auto-checkers only check return values — they ignore `print()` output.**

Example of mistake:

```
def compute_sum(arr):

    print(np.sum(arr))  # Wrong: nothing returned to test
```

Fix:
✅ **Must return value:**

```
def compute_sum(arr):
    return np.sum(arr)
```

✅ **Only use print() if the problem specifically asks for it — otherwise return the result.**

✅ **Checklist Before You Submit:**

🔍 **What to Check**

✅ **Function name is exactly as given?**

✅ **Return type matches problem (np.ndarray, tuple, float)?**

✅ **No hardcoded values?**

✅ **Used np.mean(), np.sum(), np.where() correctly?**

✅ No random values unless asked?

✅ Empty input validated (if asked)?

✅ Used np.round() or correct formatting as required?

✅ No print statements instead of return?

✅ Tested function manually before final submission?

🚨 **Golden Rule:**

"Write for auto-checker, not for your eyes."
👉 **Print ≠ Return**
👉 **Predictable output only**
👉 **Follow exactly as question says — nothing extra, nothing missing.**

Why Randomization & Hardcoding Errors Cause Test Case Failures (Even If Logic is Correct)

Your logic may work, output may "look fine" — but test cases expect fixed, predictable output — not something random.

Hardcoding Error

Why does this happen?

- You manually wrote the "expected result" instead of using logic or NumPy functions.
- Your output **matches the test sample** but fails when a new test input is given.

🎯 Golden Rule for All Students:

> **"Test cases are like strict judges — they only pass code that follows the question exactly, not what looks okay on your screen."**

✅ If you follow this — 100% of NumPy Assessment test cases will pass.

🎯 Real Engineers don't rely on "It works on my machine" — they make it work for every machine, every test case, every input.

That's why these small mistakes are actually **golden lessons**. You are learning what most freshers don't learn until their first job.

🌱 Don't stop. Don't skip. Don't fear test failures.

Every failure today is **saving you from a bigger failure in your job** tomorrow.
Every error message is a **free lesson that makes you sharper**.

💡 **And remember:**
✔️ You are not slow.

✔️ You are learning something 90% of non-programmers cannot do.
✔️ You are closer to being a real Data Engineer than you think.

🔥 Keep Going — Success in Tech is for the Persistent.

"First, it confuses you. Then it clicks. Then it becomes your second nature."

I believe in you.
Your trainers believe in you.
Soon, your code will pass **every test case — not by luck, but because you mastered it.**

🚀 Code. Break. Fix. Win.

You've got this. 💪

Assessment: Health Tracker System using NumPy

📌 Problem Statement
You are asked to design a Health Tracker System to process daily step counts of users using NumPy arrays. The system should validate the data, process it, categorize the step counts, assign grade labels, find streaks, and compute statistics.

📌 Class:

class HealthTrackerAnalyzer:

📌 Operations to Implement:

1. Create Daily Steps Array
Function:
def create_steps_array(self, steps_list: list) -> np.ndarray:

Input Example:
create_steps_array([5000, 7000, 3000, 10000, 8000])
Expected Output:
array([5000, 7000, 3000, 10000, 8000])
2. Validate Steps Array
Function:
def validate_steps_array(self, steps_array: np.ndarray) -> bool:

Description:
Non-empty

Numeric

All values >= 0

3. Add New Day's Step Count
Function:
def add_new_day_steps(self, steps_array: np.ndarray, new_steps: int) -> np.ndarray:
Description:
Append the new day's step count to the existing array.

4. Categorize Daily Activity
Function:
def categorize_daily_activity(self, steps_array: np.ndarray) -> np.ndarray:
Description:
≥ 8000: "Active"

5000–7999: "Moderate"

< 5000: "Sedentary"

Example Output:
array(['Moderate', 'Moderate', 'Sedentary', 'Active', 'Active'], dtype='<U9')
5. Assign Grade Notation
Function:
def assign_grade_notation(self, steps_array: np.ndarray) -> np.ndarray:
Description:
≥ 9000: "A"

7000–8999: "B"

5000–6999: "C"

< 5000: "D"

Example Output:
array(['C', 'B', 'D', 'A', 'B'], dtype='<U1')

6. Update Step Count for a Specific Day
Function:

def update_steps_for_day(self, steps_array: np.ndarray, day_index: int, new_steps: int) -> np.ndarray:

Description:
Updates the value at the specified index (day).

7. Get Days with High Activity (Steps > Threshold)

Function:
def get_high_activity_days(self, steps_array: np.ndarray, threshold: int) -> list:
Description:
Returns a list of step counts where steps > threshold.

8. Format Steps into Strings

Function:
def format_steps_as_strings(self, steps_array: np.ndarray) -> np.ndarray:
Description:
Returns a string array like: ["5000 steps", "7000 steps", ...]

9. Find Longest Active Streak

Function:
def find_longest_active_streak(self, steps_array: np.ndarray, threshold: int) -> int:
Description:
Longest consecutive days with steps ≥ threshold.

Example:
find_longest_active_streak(np.array([8000, 9000, 4000, 10000, 11000]), 8000) # Output: 2

10. Compute Statistics

Function:
def compute_statistics(self, steps_array: np.ndarray) -> tuple:
Description:
Returns (mean, min, max, standard deviation) rounded to 2 decimals.

Example Output:
(6600.0, 3000, 10000, 2529.82)
🎯 Student Deliverable:
Implement HealthTrackerAnalyzer class

Notes :

Validate array input properly

Use NumPy functions for efficient operations

Return exact expected output formats.
 Additional Task in Numpy:

1. Convert NumPy Array to List
Function:

```
def convert_array_to_list(self, steps_array: np.ndarray) -> list:
```

Description:
Convert the NumPy array to a list and return.

2. Get Total Steps from List
Function:

```
def total_steps_from_list(self, steps_list: list) -> int:
```

Description:
Calculate the total steps using the list.

3. Create Tuple of (Min Steps, Max Steps)
Function:

```
def min_max_steps_tuple(self, steps_list: list) -> tuple:
```

Description:
Return a tuple: (minimum steps, maximum steps) from the list.

4. Filter Active Days Using List Comprehension
Function:

```
def filter_active_days(self, steps_list: list, threshold: int) -> list:
```

Description:

Return a new list containing only the days where steps ≥ threshold.

5. Format Step Values in List as Strings
Function:

```
def format_steps_list_as_strings(self, steps_list: list) -> list:
```

Description:
Return a list of strings like ["5000 steps", "7000 steps", ...].

Numpy 1:

```
class StockAnalyzer:
    def create_stock_array(self, stocks):
        return np.array(stocks)

    def validate_stocks(self, arr):
        return arr.size > 0 and np.all(arr >= -100) and np.all(arr <= 100)

    def compute_stocks(self, arr):
        return round(arr.mean(), 2), round(arr.std(), 2), round(arr.max(), 2)

    def flag_stocks(self, arr):
        return np.where(arr > 5, 'High Risk', np.where(arr > 2, 'Moderate Risk', 'Stable'))

    def longest_loss_streak(self, arr):
        streak = 0
        max_streak = 0
        for val in arr:
            if val < 0:
                streak += 1
                max_streak = max(max_streak, streak)
            else:
                streak = 0
        return max_streak
```

Problem Set 1: Stock Price Analyzer

1. **Create Stock Price Change Array**
   create_stock_array(price_changes: list) -> np.ndarray

2. **Compute Volatility Metrics**
   compute_volatility(stock_array: np.ndarray) -> tuple

3. **Flag Volatile Stocks**
   flag_volatile_stocks(stock_array: np.ndarray) -> np.ndarray

4. **Identify Consecutive Loss Days**
   longest_loss_streak(stock_array: np.ndarray) -> int

Temperature Analyzer

---

1. Create Temperature Array

Function:

1. **def create_temperature_array(temperatures: list) -> np.ndarray:**

Description:
Creates a NumPy array from a list of temperature readings.

Example Input:

2. **create_temperature_array([36.5, 37.2, 38.0, 36.8, 39.1])**

Expected Output:

3. array([36.5, 37.2, 38.0, 36.8, 39.1])

---

2. Validate Temperature Array

Function:

4. def validate_temperature_array(temperature_array: np.ndarray) -> bool:

Description:
Checks if the temperature values are between 30°C and 45°C (valid human body temperature range).

Example Input:

5. validate_temperature_array(np.array([36.5, 37.2, 50.0]))

Expected Output:

6. False

---

3. Compute Temperature Metrics

Function:

7. **def compute_temperature_metrics(temperature_array: np.ndarray) -> tuple:**

Description:
Returns average temperature, maximum temperature, and minimum temperature.

Example Input:

8. **compute_temperature_metrics(np.array([36.5, 37.2, 38.0, 36.8, 39.1]))**

Expected Output:

9. **(37.12, 39.1, 36.5)**

---

4. Detect Abnormal Temperatures

Function:

10. **def detect_abnormal_temperatures(temperature_array: np.ndarray) -> np.ndarray:**

Description:
Flags temperatures as:

- **"Hypothermia" if < 35°C**
- **"Normal" if 35°C–37.5°C**
- **"Fever" if > 37.5°C**

Example Input:

11. detect_abnormal_temperatures(np.array([34.5, 36.6, 38.2]))

Expected Output:

12. ['Hypothermia', 'Normal', 'Fever']

---

5. Identify Longest Normal Temperature Streak

Function:

13. def longest_normal_temperature_streak(temperature_array: np.ndarray) -> int:

Description:
Returns the longest consecutive streak where temperature was in the normal range (35°C–37.5°C).

Example Input:

14. longest_normal_temperature_streak(np.array([34.5, 36.6, 36.8, 38.2, 36.5, 36.4]))

Expected Output:

15. 3

---

6. Format Temperature Readings

Function:

16. def format_temperature_readings(temperature_array: np.ndarray) -> np.ndarray:

Description:
Formats temperatures as strings with " °C" and two decimal places.

Example Input:

17. format_temperature_readings(np.array([36.567, 38.234]))

Expected Output:

18. ['36.57 °C', '38.23 °C']

Summary:

Heart Rate Analyzer	Temperature Analyzer
Heart Rate (BPM) array	Temperature (°C) array
Validate: 40–180 BPM	Validate: 30–45 °C
Metrics: avg, max, min	Same Metrics
Abnormal: Brady/Tachy	Abnormal: Hypo/Fever
Longest "Normal" Streak	Longest Normal Temp Streak
Format: " BPM"	Format: " °C"

Problem Statement: Fuel Consumption Analyzer

A transport company maintains records of fuel consumption (in litres) for its fleet vehicles. You need to design a system that processes this fuel data, validates entries, computes fuel statistics, adjusts values based on discounts for bulk fuel usage, flags heavy fuel-consuming trips, and formats results for reporting.

📌 Class Creation

```
class FuelDataAnalyzer:
    def __init__(self):
        pass
```

📌 Operations

---

1. Create Fuel Consumption Array

Function Prototype:

```
def create_fuel_array(self, fuel_list: list) -> np.ndarray:
```

Description:
Converts a list of fuel consumption amounts (in litres) into a NumPy array.

Example Input:

```
create_fuel_array([50.5, 120.0, 75.3, 99.99])
```

Expected Output:

```
array([50.5, 120.0, 75.3, 99.99])
```

---

2. Validate Fuel Array

Function Prototype:

```
def validate_fuel_array(self, fuel_array: np.ndarray) -> bool:
```

Description:
Ensures all entries are numeric and non-negative.

Example Input:

```
validate_fuel_array(np.array([60, -10, 80]))
```

Expected Output:

```
False
```

---

3. Compute Fuel Summary

Function Prototype:

```
def compute_fuel_summary(self, fuel_array: np.ndarray) -> tuple:
```

Description:
Computes total fuel used, average per trip, and maximum fuel for any single trip.

Example Input:

```
compute_fuel_summary(np.array([50.5, 120.0, 75.3, 99.99]))
```

Expected Output:

```
(345.79, 86.4475, 120.0)
```

---

4. Apply Discount on Bulk Fuel Usage

Function Prototype:

```
def apply_bulk_discount(self, fuel_array: np.ndarray) -> np.ndarray:
```

Description:
For trips consuming more than 100 litres, apply a 10% discount (adjusted usage).

Example Input:

```
apply_bulk_discount(np.array([80, 120, 150]))
```

Expected Output:

```
array([80.0, 108.0, 135.0])
```

---

5. Flag Heavy Consumption Trips

Function Prototype:

```
def flag_heavy_consumption(self, fuel_array: np.ndarray) -> np.ndarray:
```

Description:
Label trips as "High" if consumption exceeds the average, else "Normal".

Example Input:

```
flag_heavy_consumption(np.array([50, 120, 80]))
```

Expected Output:

```
["Normal", "High", "Normal"]
```

---

6. Format Fuel Records

Function Prototype:

```
def format_fuel_readings(self, fuel_array: np.ndarray) -> np.ndarray:
```

Description:
Format fuel amounts to string like "XX.XX Litres".

Example Input:

```
format_fuel_readings(np.array([80, 120.456]))
```

Expected Output:

```
["80.00 Litres", "120.46 Litres"]
```

Summary:

Order Data Analyzer	Fuel Data Analyzer (New)
Order Amount (USD)	Fuel Consumption (Litres)
apply_discount	apply_bulk_discount
compute_order_summary	compute_fuel_summary

flag_high_value_orders	**flag_heavy_consumption**
format_order_amounts	**format_fuel_readings**