Python: Functions & Modularity (Subjective)

Python: Functions & Modularity

Question

You are building a **mini student grade analyzer** for a classroom system that processes student scores, evaluates performance, and generates a summary report.

Task 1 — Process the Scores Write a function `process_scores(students)` that accepts a dictionary where keys are student names and values are lists of integer scores. Calculate the average score for each student (rounded to 2 decimal places) and return a new dictionary in the format `{ name: average_score }`.

Task 2 — Classify the Grades Write a function `classify_grades(averages)` that takes the output of Task 1 and assigns a letter grade to each student using the scale below. Return a dictionary in the format `{ name: (average, grade) }`. Define the grading thresholds as variables **inside** the function — do not use global variables for them.

Average	Grade
90 and above	A
75 – 89	B
60 – 74	C
Below 60	F

Task 3 — Generate the Report Write a function `generate_report(classified, passing_avg=70)` that takes the output of Task 2 and an optional passing threshold (default `70`). Print a formatted report as shown below, then return the total number of students who passed. In your `main` block, call all three functions in sequence, each feeding its output into the next.

```
===== Student Grade Report =====
Alice     | Avg: 86.25 | Grade: B | Status: PASS
Bob       | Avg: 62.50 | Grade: C | Status: PASS
Clara     | Avg: 96.25 | Grade: A | Status: PASS
================================
Total Students : 3
Passed         : 3
Failed         : 0
```

Submission Guidelines

- Name your file `grade_analyzer.py` or `grade_analyzer.ipynb`
- Upload to **GitHub** (public repository) or **Google Drive** (anyone with the link → Viewer)
- Paste the public link in the assignment form before the deadline
- Verify your link works in an incognito window before submitting — inaccessible submissions will not be graded

---

Step 1 — Plan the data flow first

Before writing any code, trace how data moves across the three functions. Each function does one job and passes its result forward.

```
students dict  →  process_scores()  →  averages dict
averages dict  →  classify_grades() →  classified dict
classified dict → generate_report() →  printed report + pass count
```

Step 2 — Write `process_scores()`

Loop through each student, compute their average using `sum()` and `len()`, round it, and store it in a new dictionary.

```
def process_scores(students):
    averages = {}
    for name, scores in students.items():
        averages[name] = round(sum(scores) / len(scores), 2)
    return averages
```

`students` is a **parameter** — it only exists inside this function. We return a brand-new dictionary without touching the original data.

Step 3 — Write `classify_grades()`

Define the thresholds as a **local variable** inside the function so nothing outside can accidentally alter the grading rules. Then loop through the averages and assign a letter grade using `if/elif/else`.

```
def classify_grades(averages):
    thresholds = {"A": 90, "B": 75, "C": 60}
```

```
    classified = {}

    for name, avg in averages.items():
        if avg >= thresholds["A"]:
            grade = "A"
        elif avg >= thresholds["B"]:
            grade = "B"
        elif avg >= thresholds["C"]:
            grade = "C"
        else:
            grade = "F"
        classified[name] = (avg, grade)

    return classified
```

`thresholds` is a **local variable** — it is created when the function is called and destroyed when it returns. This is **local scope** in action, keeping the logic self-contained.

Step 4 — Write `generate_report()`

Use `passing_avg=70` as a **default argument** so the caller can override it if needed. Unpack each tuple with `for name, (avg, grade)` and track the pass count locally before returning it.

```
def generate_report(classified, passing_avg=70):
    print("===== Student Grade Report =====")
    passed = 0

    for name, (avg, grade) in classified.items():
        status = "PASS" if avg >= passing_avg else "FAIL"
        if status == "PASS":
            passed += 1
        print(f"{name:<10}| Avg: {avg:<6} | Grade: {grade} | Status: {status}")

    total = len(classified)
    print("================================")
    print(f"{'Total Students':<15}: {total}")
    print(f"{'Passed':<15}: {passed}")
    print(f"{'Failed':<15}: {total - passed}")

    return passed
```

`passed` is a **local variable** that accumulates state and is then returned. The function both prints a side effect and returns a useful value — a clean, reusable design.

Step 5 — Wire everything together in `main`

```
if __name__ == "__main__":
    students = {
        "Alice": [85, 92, 78, 90],
        "Bob":   [60, 55, 70, 65],
        "Clara": [95, 98, 100, 92]
    }

    averages     = process_scores(students)
    classified   = classify_grades(averages)
    total_passed = generate_report(classified)

    print(f"\nFinal Count — Students who passed: {total_passed}")
```

`if __name__ == "__main__"` ensures this block only runs when the file is executed directly, not when imported by another script — a standard Python best practice.

Complete Solution

```
def process_scores(students):
    averages = {}
    for name, scores in students.items():
        averages[name] = round(sum(scores) / len(scores), 2)
    return averages


def classify_grades(averages):
    thresholds = {"A": 90, "B": 75, "C": 60}
    classified = {}
    for name, avg in averages.items():
        if avg >= thresholds["A"]:
            grade = "A"
        elif avg >= thresholds["B"]:
            grade = "B"
        elif avg >= thresholds["C"]:
            grade = "C"
        else:
            grade = "F"
        classified[name] = (avg, grade)
    return classified


def generate_report(classified, passing_avg=70):
    print("===== Student Grade Report =====")
    passed = 0
    for name, (avg, grade) in classified.items():
        status = "PASS" if avg >= passing_avg else "FAIL"
        if status == "PASS":
            passed += 1
```

```
        print(f"{name:<10}| Avg: {avg:<6} | Grade: {grade} | Status: {status}")
    total = len(classified)
    print("================================")
    print(f"{'Total Students':<15}: {total}")
    print(f"{'Passed':<15}: {passed}")
    print(f"{'Failed':<15}: {total - passed}")
    return passed


if __name__ == "__main__":
    students = {
        "Alice": [85, 92, 78, 90],
        "Bob":   [60, 55, 70, 65],
        "Clara": [95, 98, 100, 92]
    }
    averages     = process_scores(students)
    classified   = classify_grades(averages)
    total_passed = generate_report(classified)
    print(f"\nFinal Count — Students who passed: {total_passed}")
```

Python: Files & JSON - Subjective

Working with Files and JSON in Python

You are helping a small bookstore save and manage their inventory. The store currently has a list of books stored in a file called `inventory.json`. Your job is to read the inventory, make changes, and save it back — all using proper Python file-handling practices.

You are given the following starter file `inventory.json`:

```
[
  {"title": "The Alchemist", "author": "Paulo Coelho", "price": 12.99, "in_stock":
  {"title": "1984", "author": "George Orwell", "price": 9.99, "in_stock": true}
]
```

And the following new book to add:

```
new_book = {"title": "Atomic Habits", "author": "James Clear", "price": 14.99, "in
```

Task 1 — Read the inventory Open `inventory.json` using a `with` block and load its contents into a variable called `inventory`. Print the total number of books currently in the file.

Task 2 — Update and save Append `new_book` to the `inventory` list. Write the updated list back to `inventory.json` using a `with` block, with an indentation of 4 spaces.

Task 3 — Display the inventory Read the updated file again and print each book's details in the following format:

```
Title: The Alchemist | Author: Paulo Coelho | Price: $12.99
Title: 1984 | Author: George Orwell | Price: $9.99
Title: Atomic Habits | Author: James Clear | Price: $14.99
```

Submission Guidelines

- Write your code along with explanations in a Word file or Google Doc.
- Submit using Google Drive or GitHub only.

Google Drive:

- Set access to "Anyone with the link – Viewer."
- Ensure the link opens without access requests.

GitHub:

- Upload files to a public repository.
- Ensure all required files are pushed and accessible.

Private links or inaccessible submissions will not be evaluated.

Task 1 — Read the inventory

```
import json

with open('inventory.json', 'r') as f:
    inventory = json.load(f)

print(f"Total books: {len(inventory)}")
```

Step-by-step:

- `import json` loads Python's built-in module for working with JSON data.
- `open('inventory.json', 'r')` opens the file in **read mode** (`'r'`).
- The `with` block acts as a context manager — it automatically closes the file once the block is done, even if an error occurs. You don't need to call `f.close()` manually.
- `json.load(f)` reads the file and converts the JSON array into a Python **list of**

- `json.load(f)` reads the file and converts the JSON array into a Python **list of dictionaries**, stored in `inventory`.
- `len(inventory)` counts the number of items in the list, which is `2` at this point.

Output:

```
Total books: 2
```

Task 2 — Update and save

```
new_book = {"title": "Atomic Habits", "author": "James Clear", "price": 14.99, "in

inventory.append(new_book)

with open('inventory.json', 'w') as f:
    json.dump(inventory, f, indent=4)
```

Step-by-step:

- `inventory.append(new_book)` adds the new book dictionary to the **end** of the list in memory. The file is not touched yet at this point.
- `open('inventory.json', 'w')` opens the file in **write mode** (`'w'`). This clears the existing file content so the full updated list can be written fresh.
- `json.dump(inventory, f, indent=4)` converts the Python list back into JSON and writes it directly into the file. The `indent=4` argument makes the output neatly formatted with 4-space indentation, which is easier for humans to read.

> 💡 **Key difference to remember:**
>
> - `json.dump()` → writes to a **file**
> - `json.dumps()` → converts to a **string** (the `s` stands for *string*)

Task 3 — Display the inventory

```
with open('inventory.json', 'r') as f:
    updated_inventory = json.load(f)

for book in updated_inventory:
    print(f"Title: {book['title']} | Author: {book['author']} | Price: ${book['pri
```

Step-by-step:

- The file is opened again with a fresh `with` block to confirm the data was saved correctly

- The file is opened again with a fresh `with` block to confirm the data was saved correctly.
- `json.load(f)` loads the updated JSON array back into `updated_inventory` as a Python list.
- The `for` loop goes through each dictionary in the list one by one.
- The f-string uses `book['title']`, `book['author']`, and `book['price']` to pull out only the fields we need from each dictionary and format them into the required output.

Output:

```
Title: The Alchemist | Author: Paulo Coelho | Price: $12.99
Title: 1984 | Author: George Orwell | Price: $9.99
Title: Atomic Habits | Author: James Clear | Price: $14.99
```

Complete Solution

```
import json

new_book = {"title": "Atomic Habits", "author": "James Clear", "price": 14.99, "in

# Task 1: Read the inventory
with open('inventory.json', 'r') as f:
    inventory = json.load(f)

print(f"Total books: {len(inventory)}")

# Task 2: Append new book and save
inventory.append(new_book)

with open('inventory.json', 'w') as f:
    json.dump(inventory, f, indent=4)

# Task 3: Read back and display
with open('inventory.json', 'r') as f:
    updated_inventory = json.load(f)

for book in updated_inventory:
    print(f"Title: {book['title']} | Author: {book['author']} | Price: ${book['pri
```

> **Concepts Practiced:** Reading files · Writing files · `with` context managers · `json.load()` · `json.dump()` · Iterating over a list of dictionaries

Python: Loops & Automation - Subjective

Question: Weekly Temperature Analysis

Write a Python program to analyze temperature data for one week.

Task 1: Find Maximum and Minimum

Write code to find the highest and lowest temperature from the list.

Input:

```
temperatures = [28, 32, 35, 29, 31, 27, 30]
```

Expected Output:

```
Highest Temperature: 35°C
Lowest Temperature: 27°C
```

Requirements:

- Use a `for` loop to check each temperature
- Do NOT use built-in functions like `max()` or `min()`

Task 2: Count Hot Days

Count how many days had temperature above 30°C. Skip days with temperature ≤30°C using `continue`.

Input:

```
temperatures = [28, 32, 35, 29, 31, 27, 30]
```

Expected Output:

```
Hot Days (>30°C): 3
```

Requirements:

- Use a `for` loop
- Use `continue` to skip days that are not hot (≤30°C)

Task 3: Alert System

Count hot days but stop immediately if temperature reaches 40°C or higher using `break`.

Input:

```
temperatures = [28, 32, 35, 40, 31, 33, 30]
```

Expected Output:

```
Hot Days before alert: 2
Alert! Extreme temperature 40°C detected on Day 4
```

Requirements:

- Use a `for` loop with day counter
- Use `break` when temperature ≥ 40°C
- Count only hot days (>30°C) before the alert

Submission Guidelines

File Structure:

Create a file named: `temperature_analysis.py`

```
# Name: [Your Name]
# Roll Number: [Your Roll Number]
# Assignment: Python Loops & Automation – Subjective Question

print("===== Task 1: Find Maximum and Minimum =====")
temperatures = [28, 32, 35, 29, 31, 27, 30]
# Write your code here


print("\n===== Task 2: Count Hot Days =====")
temperatures = [28, 32, 35, 29, 31, 27, 30]
# Write your code here


print("\n===== Task 3: Alert System =====")
temperatures = [28, 32, 35, 40, 31, 33, 30]
# Write your code here
```

How to Submit:

How to Submit:

Option 1: GitHub

- Create a repository named `python-loops-assignment`
- Upload your `temperature_analysis.py` file
- Share the repository link

Option 2: Google Drive

- Upload your `temperature_analysis.py` file
- Set sharing to "Anyone with the link can view"
- Share the file link

Editorial Solution

```
# Name: Solution Key
# Roll Number: N/A
# Assignment: Python Loops & Automation – Subjective Question

print("===== Task 1: Find Maximum and Minimum =====")
temperatures = [28, 32, 35, 29, 31, 27, 30]

# Initialize with first temperature
highest = temperatures[0]
lowest = temperatures[0]

# Loop through all temperatures
for temp in temperatures:
    if temp > highest:
        highest = temp
    if temp < lowest:
        lowest = temp

print(f"Highest Temperature: {highest}°C")
print(f"Lowest Temperature: {lowest}°C")


print("\n===== Task 2: Count Hot Days =====")
temperatures = [28, 32, 35, 29, 31, 27, 30]

hot_days = 0

# Loop through temperatures
for temp in temperatures:
    if temp <= 30:
        continue  # Skip this day, not hot
    hot_days += 1  # Count as hot day

print(f"Hot Days (>30°C): {hot_days}")
```

```
print("\n===== Task 3: Alert System =====")
temperatures = [28, 32, 35, 40, 31, 33, 30]

hot_days = 0
day_number = 0

# Loop through temperatures
for temp in temperatures:
    day_number += 1

    # Check for extreme heat
    if temp >= 40:
        print(f"Hot Days before alert: {hot_days}")
        print(f"Alert! Extreme temperature {temp}°C detected on Day {day_number}")
        break  # Stop immediately

    # Count hot days
    if temp > 30:
        hot_days += 1
```

Detailed Explanation

Task 1 Walkthrough:

Step 1: Start with first value

```
highest = temperatures[0]  # 28
lowest = temperatures[0]   # 28
```

Step 2: Loop and compare

- Day 1: temp = 28 → no change
- Day 2: temp = 32 → highest becomes 32
- Day 3: temp = 35 → highest becomes 35
- Day 4: temp = 29 → no change
- Day 5: temp = 31 → no change
- Day 6: temp = 27 → lowest becomes 27
- Day 7: temp = 30 → no change

Result: Highest = 35, Lowest = 27

Task 2 Walkthrough:

Task 2 Walkthrough:

Loop execution:

- temp = 28 → 28 ≤ 30 → `continue` (skip)
- temp = 32 → 32 > 30 → count = 1
- temp = 35 → 35 > 30 → count = 2
- temp = 29 → 29 ≤ 30 → `continue` (skip)
- temp = 31 → 31 > 30 → count = 3
- temp = 27 → 27 ≤ 30 → `continue` (skip)
- temp = 30 → 30 ≤ 30 → `continue` (skip)

Result: Hot Days = 3

Key Point: `continue` skips the rest of the loop and jumps to next iteration.

Task 3 Walkthrough:

Loop execution:

- Day 1: temp = 28 → not ≥40, not >30 → count = 0
- Day 2: temp = 32 → not ≥40, is >30 → count = 1
- Day 3: temp = 35 → not ≥40, is >30 → count = 2
- Day 4: temp = 40 → **is ≥40** → `break` (exit loop immediately)

Days 5, 6, 7 are never checked because `break` stopped the loop.

Result: Hot Days = 2, Alert on Day 4

Key Point: `break` exits the loop completely, no further iterations happen