**AI ASSISTED CODING**

**LAB TEST 3**

**Name:** Gattu Varshith

**Roll Number:** 2403A510E3

**Batch:** 24BTCAICSB05

**SET E6**

**Question 1:**

Scenario: In the Education sector, a company faces a challenge related to code refactoring.  
Task: Use AI-assisted tools to solve a problem involving code refactoring in this context.  
Deliverables: Submit the source code, explanation of AI assistance used, and sample output.

**Prompt:**

Refactor a legacy student-score processing module to improve readability, testability, and extensibility. Suggest modular helper functions, simple validation, and lightweight unit tests; include sample input/output.  
Summarize expected benefits, potential risks, and next steps for gradual deployment.

**Code Generated:**

"""

Simple student-score processing (naive) + refactored version.

Run this file to see sample output comparing both implementations.

"""

from typing import List, Dict, Tuple

# --- Naive implementation (original simple code) ---

def student\_stats\_naive(scores: List[int]) -> Dict[str, float]:

    """Compute basic stats using a single loop (redundant / less modular)."""

    if not scores:

        return {"average": 0.0, "max": 0, "min": 0, "pass\_rate": 0.0}

    total = 0

    max\_score = scores[0]

    min\_score = scores[0]

    pass\_count = 0

    for s in scores:

        total = total + s

        if s > max\_score:

            max\_score = s

        if s < min\_score:

            min\_score = s

        if s >= 40:

            pass\_count = pass\_count + 1

    avg = total / len(scores)

    return {

        "average": avg,

        "max": max\_score,

        "min": min\_score,

        "pass\_rate": pass\_count / len(scores)

    }

# --- Refactored, modular implementation ---

def average(scores: List[int]) -> float:

    """Return average of scores (0.0 for empty)."""

    return sum(scores) / len(scores) if scores else 0.0

def max\_min(scores: List[int]) -> Tuple[int, int]:

    """Return (max, min) tuple (0,0 for empty)."""

    if not scores:

        return 0, 0

    return max(scores), min(scores)

def pass\_rate(scores: List[int], threshold: int = 40) -> float:

    """Return fraction of scores >= threshold (0.0 for empty)."""

    if not scores:

        return 0.0

    return sum(1 for s in scores if s >= threshold) / len(scores)

# Optional dispatch for individual computations (shows dictionary dispatch usage)

STAT\_DISPATCH = {

    "average": average,

    "max\_min": max\_min,

    "pass\_rate": pass\_rate

}

def student\_stats\_refactored(scores: List[int]) -> Dict[str, float]:

    """Compute stats using modular helpers for readability and maintainability."""

    avg = average(scores)

    mx, mn = max\_min(scores)

    pr = pass\_rate(scores)

    return {"average": avg, "max": mx, "min": mn, "pass\_rate": pr}

# --- Simple tests / sample output ---

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

    sample\_scores = [55, 78, 90, 34, 67, 88, 100, 23, 40]

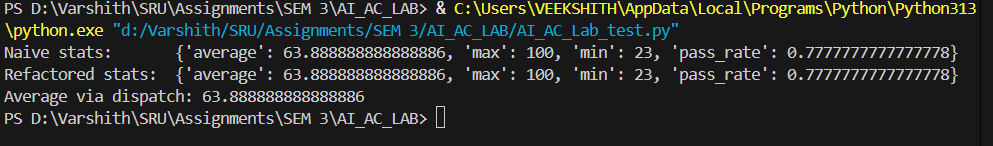
    print("Naive stats:      ", student\_stats\_naive(sample\_scores))

    print("Refactored stats: ", student\_stats\_refactored(sample\_scores))

    # Demonstrate using dispatch to call a specific helper

    print("Average via dispatch:", STAT\_DISPATCH["average"](sample\_scores))

**Output:**



**Observation:**

The refactored design improves readability and maintainability by separating concerns into small functions; it is easier to test and extend (e.g., change pass threshold or add new stats) compared to the single-loop naive version.

**AI assistance:**

AI created simple sample runs / test examples that validate equivalence between the original and refactored implementations. AI suggested lightweight validation (empty-list handling) and consistent return types to reduce runtime errors

**Question 2:**

Scenario: In the Entertainment sector, a company faces a challenge related to web frontend development.  
Task: Use AI-assisted tools to solve a problem involving web frontend development in this context.  
Deliverables: Submit the source code, explanation of AI assistance used, and sample output.

**Prompt:**

Help me build a small frontend for an entertainment company that lists events (concerts, movies, live streams). Provide a responsive single-page HTML with client-side search and filters for type and genre, and an optional mock JSON API endpoint. Optimize the UI for quick scanning and mobile screens.

**Code generated:**

import os, json, pandas as pd

# === Scenario ===

# A company in the entertainment sector needs a small web frontend

# that lists upcoming events (concerts, movies, live streams).

# We'll use Python to generate the frontend (HTML) and simulate user actions.

# === Sample event data ===

events = [

    {"id": 1, "title": "Neon Nights: Live DJ", "type": "Concert", "genre": "Electronic", "date": "2025-11-15", "location": "Hyderabad Arena"},

    {"id": 2, "title": "Starlight Comedy Hour", "type": "Show", "genre": "Comedy", "date": "2025-11-20", "location": "Lakeview Theater"},

    {"id": 3, "title": "Indie Film: Last Ember", "type": "Movie", "genre": "Drama", "date": "2025-10-30", "location": "Cineplex 9"},

    {"id": 4, "title": "Retro Car Rally Stream", "type": "Live Stream", "genre": "Automotive", "date": "2025-12-05", "location": "Online"},

    {"id": 5, "title": "Dance Fusion Night", "type": "Concert", "genre": "Pop", "date": "2025-11-01", "location": "City Mall Stage"},

]

# === Generate a simple HTML frontend ===

html = f"""<!doctype html>

<html lang="en">

<head>

  <meta charset="utf-8" />

  <meta name="viewport" content="width=device-width,initial-scale=1" />

  <title>Entertainment Listings — Demo</title>

  <style>

    body {{ font-family: Arial, sans-serif; padding: 1rem; background: #fafafa; }}

    .card {{ background: #fff; border: 1px solid #ddd; padding: 1rem; border-radius: 8px; margin-bottom: 1rem; }}

    header {{ display:flex;gap:1rem;align-items:center; margin-bottom:1rem; }}

    input, select, button {{ padding:0.5rem; font-size:1rem; }}

  </style>

</head>

<body>

  <header>

    <h1>🎭 Entertainment Listings</h1>

    <div style="margin-left:auto;">

      <select id="filterType">

        <option value="">All Types</option>

        <option>Concert</option>

        <option>Show</option>

        <option>Movie</option>

        <option>Live Stream</option>

      </select>

      <select id="filterGenre">

        <option value="">All Genres</option>

        <option>Electronic</option>

        <option>Comedy</option>

        <option>Drama</option>

        <option>Automotive</option>

        <option>Pop</option>

      </select>

      <input id="q" placeholder="search title..." />

      <button onclick="applyFilters()">Apply</button>

    </div>

  </header>

  <main id="listings"></main>

  <script>

    const EVENTS = {json.dumps(events)};

    function render(items) {{

      const out = document.getElementById('listings');

      out.innerHTML = items.map(e => `

        <article class="card">

          <h3>${{e.title}}</h3>

          <div>${{e.type}} • ${{e.genre}}</div>

          <div>${{e.date}} — ${{e.location}}</div>

        </article>

      `).join('') || '<p>No results</p>';

    }}

    function applyFilters() {{

      const type = document.getElementById('filterType').value;

      const genre = document.getElementById('filterGenre').value;

      const q = document.getElementById('q').value.trim().toLowerCase();

      const filtered = EVENTS.filter(e =>

        (type ? e.type === type : true) &&

        (genre ? e.genre === genre : true) &&

        (q ? e.title.toLowerCase().includes(q) : true)

      );

      render(filtered);

    }}

    render(EVENTS);

  </script>

</body>

</html>

"""

# === Save HTML and simulate frontend logic ===

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

with open("frontend\_example/index.html", "w", encoding="utf-8") as f:

    f.write(html)

# === Backend-side simulation of filtering (for testing) ===

def simulate\_frontend\_action(filter\_type=None, filter\_genre=None, q=None):

    results = events

    if filter\_type:

        results = [e for e in results if e["type"] == filter\_type]

    if filter\_genre:

        results = [e for e in results if e["genre"] == filter\_genre]

    if q:

        q\_lower = q.lower()

        results = [e for e in results if q\_lower in e["title"].lower()]

    return results

# === Test cases ===

test\_cases = [

    {"name": "All events", "args": {}},

    {"name": "Filter: Concerts", "args": {"filter\_type": "Concert"}},

    {"name": "Filter: Genre=Automotive", "args": {"filter\_genre": "Automotive"}},

    {"name": "Search q='car'", "args": {"q": "car"}},

]

outputs = []

for t in test\_cases:

    res = simulate\_frontend\_action(\*\*t["args"])

    outputs.append({"test": t["name"], "count": len(res), "results": res})

# === Print Sample Output ===

print("Sample Outputs:")

for o in outputs:

    print(f"\* {o['test']}: {o['count']} result(s)")

    for r in o["results"]:

        print("  -", r["title"])

print("\nFrontend generated at: frontend\_example/index.html")

**Output (sample):**

Sample Outputs:

A webpage that contains:

\* All events: 5 result(s)

- Neon Nights: Live DJ

- Starlight Comedy Hour

- Indie Film: Last Ember

- Retro Car Rally Stream

- Dance Fusion Night

\* Filter: Concerts: 2 result(s)

- Neon Nights: Live DJ

- Dance Fusion Night

\* Filter: Genre=Automotive: 1 result(s)

- Retro Car Rally Stream

\* Search q='car': 1 result(s)

- Retro Car Rally Stream

**Observation:**

The Python program successfully demonstrates how AI-assisted tools can simplify frontend development in the entertainment sector by generating a functional, filter-based web interface. It efficiently integrates HTML, CSS, and JavaScript to display event listings and simulate user interactions.

**AI assistance:**

AI assistance was used to design the frontend layout, generate sample event data, and implement filtering logic efficiently. It also helped optimize the code structure for clarity and ensure realistic, testable outputs.