University of Engineering and Technology, Taxila

Department of Computer Engineering

Lab Report - 01

Course – Programming for AI

Submitted by – Kinza Habib

Registration no – 25-MS-AI-11

Submitted to – Dr Naveed Khan

Submission Date – Nov 6, 2025

Objective

The purpose of this lab is to build a strong foundation in **Python programming**, which is essential for Artificial Intelligence and Machine Learning.
This lab covers variables, operators, strings, conditionals, loops, functions, data structures, and basic file handling.

Component	Details
Python Version	3.11.0
Operating System	Windows 10 (64-bit)
Editor	Visual Studio Code
Virtual Environment	.venv
Built-in Modules Used	math, csv, json, re, functools
External Libraries	None

A1) Hello & Variables

Code Summary:

This task demonstrates how to create and use variables in Python. Two variables name (string) and age (integer) are defined and printed using the print() function. The type() function is also used to check their data types.

```
# A1) Hello & Variables
name = "Kinza"
age = 22
print("Hello,", name, "— age:", age)
print(type(name), type(age))
```

✓ 0.0s

```
Hello, Kinza — age: 22
<class 'str'> <class 'int'>
```

Explanation:

- name stores a string value and age stores an integer.
- print() displays the values on screen.
- type() confirms that name is of type *string* and `age* is an *integer*.

A2) Arithmetic & Operators

Code Summary:
This task demonstrates the use of **arithmetic** and **bitwise** operators in Python using two variables `a` and `b`.

```
# A2) Arithmetic & operators
a, b = 15, 4
print("add:", a+b, "sub:", a-b, "mul:", a*b, "div:", a/b, "floor:", a//b, "mod:", a%b, "pow:", a**b)
print("bitwise AND:", a & b, "OR:", a | b, "XOR:", a ^ b, "shift left:", a << 1)
```

```
add: 19 sub: 11 mul: 60 div: 3.75 floor: 3 mod: 3 pow: 50625
bitwise AND: 4 OR: 15 XOR: 11 shift left: 30
```

Explanation:

- Arithmetic operators (`+`, `-`, `*`, `/`, `//`, `%`, `**`) perform basic mathematical operations.
- Bitwise operators (`&`, `|`, `^`, `<<`) act on the binary representation of integers.
- For example, `15 & 4` performs a bitwise AND operation, and `a << 1` shifts bits of `a` one place to the left (multiplies by 2).

A3) Strings: Slicing & Methods

This task shows string manipulation techniques such as trimming, lowercasing, slicing, and checking membership.

```
# A3) Strings: slicing & methods
s = "  Machine Learning  "
print(s.strip().lower())        # remove spaces, lowercase
print(s[2:9])                   # slice
print("learn" in s.lower())     # membership
print("-".join(["ai","ml","dl"]))
```

```
machine learning
Machine
True
ai-ml-dl
```

Explanation:

- `strip()` removes extra spaces; `lower()` converts to lowercase.
- `s[2:9]` extracts a substring.

- The in operator checks if a word exists in the string.
- "join" combines list elements into one string separated by -.

A4) f-Strings (Formatting)

This task demonstrates formatted output using f-strings.

```
# A4) f-strings (formatting)
item, price, qty = "book", 249.99, 3
print(f"Item: {item}, total: ₹{price*qty:,.2f}")
```

```
Item: book, total: ₹749.97
```

Explanation:

- f-strings embed variables inside strings using {}.
- :,.2f formats numbers with commas and two decimal places.
- The result displays a readable total price.

A5) If / Elif / Else

This task uses conditional statements to assign a grade based on a numeric score.

```
# A5) If/elif/else
score = 78
if score >= 90: grade = "A"
elif score >= 75: grade = "B"
elif score >= 60: grade = "C"
else: grade = "D"
print("Grade:", grade)
```

```
Grade: B
```

Explanation:

- if-elif-else checks conditions in sequence.
- Only the first true condition executes.

- The program prints the grade based on the score.

A6) Loops & Range

This task demonstrates the use of for and while loops along with the continue statement.

```
# A6) Loops & range
total = 0
for i in range(1, 6):   # 1..5
    total += i
print("sum 1..5 =", total)

n = 5
while n > 0:
    if n == 3:
        n -= 1
        continue
    print("n:", n)
    n -= 1
```

```
sum 1..5 = 15
n: 5
n: 4
n: 2
n: 1
```

Explanation:

- The for loop calculates the sum of numbers from 1 to 5.
- The while loop counts down from 5, skipping n = 3 using continue.

A7) Lists & Common Methods

This task shows how to manipulate lists by adding, removing, sorting, and slicing elements.

```python
# A7) Lists & common methods
nums = [5, 2, 9, 1]
nums.append(7); nums.remove(2); nums.sort()
print(nums)                # [1,5,7,9]
print(nums[:2], nums[-1])  # slicing and last
```

```
[1, 5, 7, 9]
[1, 5] 9
```

Explanation:

- append() adds an element, remove() deletes one.
- sort() arranges values in ascending order.
- List slicing retrieves subsets or last elements.

A8) Dictionaries & Lookup

This task demonstrates dictionary creation, key lookup, and average calculation.

```python
# A8) Dicts & lookup
student = {"name": "Aisha", "marks": [78, 82, 91]}
avg = sum(student["marks"]) / len(student["marks"])
print(student["name"], "avg =", round(avg, 1))
print(student.get("branch", "NA"))
```

```
Aisha avg = 83.7
NA
```

Explanation:

- Dictionary keys store related data.
- sum() and len() compute averages.
- get() safely retrieves a value or returns a default.

A9) Functions & Default Arguments

This task introduces defining and calling a Python function with default parameters.

```python
# A9) Functions & default args
def area_circle(r=1.0):
    from math import pi
    return pi * r * r

print(area_circle(), area_circle(2.5))
```

```
3.141592653589793 19.634954084936208
```

Explanation:

- A function is created to calculate the area of a circle.
- The default radius is 1.0 if no argument is provided.
- The math.pi constant is used for precision.

A10) Try/Except + File I/O

This task demonstrates file writing, reading, and handling exceptions.

```python
# A10) Try/except + basic file write/read
text = "hello ai\nthis is lab1"
with open("notes.txt", "w", encoding="utf-8") as f:
    f.write(text)

try:
    with open("notes.txt", "r", encoding="utf-8") as f:
        print("File contents:\n", f.read())
except FileNotFoundError:
    print("notes.txt not found")
```

```
File contents:
 hello ai
this is lab1
```

Explanation:

- The with open() block writes text into a file safely.
- The second block reads and prints file contents.
- The try/except structure prevents crashes if the file is missing.

B1) Fibonacci (Iteration & Recursion)

This task demonstrates two methods to generate the Fibonacci sequence — one using **iteration** and the other using **recursion**. The Fibonacci series is a fundamental example of sequence generation and recursion in programming.

```python
# B1) Fibonacci (Iteration & Recursion)
def fib_list(n: int):
    seq = []
    a, b = 0, 1
    for _ in range(n):
        seq.append(a)
        a, b = b, a + b
    return seq


# Recursive version
def fib(k: int) -> int:
    if k <= 1:
        return k
    return fib(k - 1) + fib(k - 2)

print("Iterative:", fib_list(10))
print("Recursive:", [fib(i) for i in range(10)])
```

```
Iterative: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Recursive: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Explanation:

- The **iterative version** builds the sequence step by step using two variables `a` and `b`.
- The **recursive version** defines the Fibonacci rule $F(n)=F(n-1)+F(n-2)$F(n) = F(n-1) + F(n-2)$F(n)=F(n−1)+F(n−2).
- Both versions generate the first 10 Fibonacci numbers:
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34.
- This exercise highlights the difference between iterative loops and recursive calls.

B2) Prime Checker & List Comprehension

This task defines a function to check whether a number is **prime** and then uses a **list comprehension** to generate all prime numbers between 2 and 50.
It demonstrates both conditional logic and Python's concise functional programming features.

```python
# 🧠 B2) Prime Checker & List Comprehension
def is_prime(x: int) -> bool:
    if x < 2:
        return False
    from math import isqrt  # integer square root
    for d in range(2, isqrt(x) + 1):
        if x % d == 0:
            return False
    return True
primes = [n for n in range(2, 50) if is_prime(n)]
print("Prime numbers from 2 to 50:")
print(primes)
```

```
Prime numbers from 2 to 50:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Explanation:

- The function is_prime() checks divisibility from 2 up to the square root of the number.
- If any divisor divides x evenly, the number is not prime.
- The **list comprehension** [n for n in range(2, 50) if is_prime(n)] collects all prime numbers in that range.
- This approach efficiently combines logic and list building in a single readable line.

B3) Word Frequency (Dictionary + Cleanup)

This task demonstrates how to clean and process text data in Python using **regular expressions** and **dictionaries**.
It counts the frequency of each word in a given sentence after converting the text to lowercase and removing punctuation.

```python
# B3) Word Frequency (Dictionary + Cleanup)
text = "AI, AI everywhere; learning AI is fun and powerful!"

import re
# Clean and split text into words
tokens = re.findall(r"[A-Za-z]+", text.lower())
# Count word frequencies using a dictionary
freq = {}
for t in tokens:
    freq[t] = freq.get(t, 0) + 1
print("Word Frequency:")
print(freq)
```

```
Word Frequency:
{'ai': 3, 'everywhere': 1, 'learning': 1, 'is': 1, 'fun': 1, 'and': 1, 'powerful': 1}
```

Explanation:

- The re.findall() function extracts only alphabetic words and ignores punctuation.
- The text.lower() ensures consistent counting by converting all words to lowercase.
- A dictionary freq stores each unique word as a key and its frequency as the value.
- The program outputs the count of each word, showing how often "AI" and other words appear.

B4) Matrix Operations with Lists

This task performs **matrix multiplication** using nested **list comprehensions** in Python.
It shows how to handle two-dimensional lists (matrices) and apply element-wise multiplication and summation to compute a result matrix.

```
# B4) Matrix Operations with Lists
A = [[1, 2, 3],
     [4, 5, 6]]

B = [[7, 8],
     [9, 10],
     [11, 12]]
# Matrix multiplication using list comprehension
C = [
    [sum(a * b for a, b in zip(Arow, Bcol)) for Bcol in zip(*B)]
    for Arow in A
]
print("Matrix C (Result of A x B):")
print(C)  # Expected: [[58, 64], [139, 154]]
```

```
Matrix C (Result of A x B):
[[58, 64], [139, 154]]
```

- zip(*B) transposes matrix **B**, allowing column-wise access.
- For each row in **A** and column in **B**, the sum(a * b for a, b in zip(Arow, Bcol)) computes the dot product.
- The final list comprehension builds the result matrix **C**.
- The output [[58, 64], [139, 154]] confirms correct matrix multiplication for a (2×3) × (3×2) operation.

B5) Set Operations (Dedupe + Overlap)

This task demonstrates how to use **Python sets** for operations such as union, intersection, and difference. Sets automatically remove duplicate elements and are ideal for comparing collections of unique items.

```python
#B5) Set Operations (Dedupe + Overlap)
a = set("machine")
b = set("learning")
# Set operations
print("Set A:", a)
print("Set B:", b)
print("Union:", a | b)     # All unique letters from both
print("Intersection:", a & b)  # Common letters
print("Only in A:", a - b)     # Letters in A but not in B
```

```
Set A: {'e', 'm', 'i', 'a', 'n', 'h', 'c'}
Set B: {'r', 'e', 'a', 'n', 'i', 'l', 'g'}
Union: {'r', 'm', 'a', 'i', 'n', 'c', 'l', 'h', 'e', 'g'}
Intersection: {'e', 'i', 'a', 'n'}
Only in A: {'c', 'h', 'm'}
```

Explanation:

- set("machine") and set("learning") convert the words into sets of unique characters.
- a | b → **Union** combines all distinct letters from both sets.
- a & b → **Intersection** gives common letters.
- a - b → **Difference** shows letters present only in *a*.
- This demonstrates how sets help in **removing duplicates** and **finding overlaps or differences** between data groups.

B6) Safe Divide with Exception Handling

This task shows how to handle **runtime errors gracefully** in Python using the try-except block. The program defines a function that performs division safely, even when the denominator is zero.

```python
#B6) Safe Divide with Exception Handling
def safe_div(a: float, b: float) -> float:
    """Safely divide a by b. Returns 'inf' if divide-by-zero occurs."""
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        print(f" Warning: Division by zero detected for {a}/{b}")
        return float("inf")  # infinity as a fallback value
print("Result 1:", safe_div(10, 2))  # normal division
print("Result 2:", safe_div(5, 0))   # triggers exception
```

```
Result 1: 5.0
 Warning: Division by zero detected for 5/0
Result 2: inf
```

Explanation:

- The function safe_div() attempts to divide a by b.
- If b is 0, a ZeroDivisionError occurs — handled by the except block.
- Instead of crashing, it prints a **warning** and returns float("inf") (infinity).
- This example demonstrates **robust programming** — preventing runtime errors from interrupting execution.

B7) Map, Filter, and Reduce Mini Example

This task demonstrates how to perform **functional-style data processing** in Python using the built-in functions map(), filter(), and reduce().

```python
#B7) Map/filter/reduce mini
from functools import reduce
nums = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x * x, nums))
# result: [1, 4, 9, 16, 25]
#filter()
evens = list(filter(lambda x: x % 2 == 0, squares))
# result: [4, 16]
# reduce()
total = reduce(lambda acc, x: acc + x, evens, 0)
# result: 20 (4 + 16)

print("Squares:", squares)
print("Evens:", evens)
print("Sum:", total)
```

```
Squares: [1, 4, 9, 16, 25]
Evens: [4, 16]
Sum: 20
```

Explanation:

- map() applies a **function** to each element of a list — here it squares each number.
- filter() selects only those elements that satisfy a **condition** — only even squares.
- reduce() combines all items into a single value by applying a function cumulatively — computes the sum of even squares.
- The output shows how these three higher-order functions can be chained together for efficient data transformations.

B8) Title-Casing Names Using String Methods

This task demonstrates how to **clean and format text data** using Python's built-in **string methods** such as strip(), split(), and capitalize().

```
#B8) Title-casing names (string methods)
raw = ["  aISha khan ", "ALI   Raza", "sara"]
def clean_name(s: str) -> str:
    return " ".join(part.capitalize() for part in s.split())
print([clean_name(r.strip()) for r in raw])
```

```
['Aisha Khan', 'Ali Raza', 'Sara']
```

Explanation:

- strip() removes **extra spaces** from the beginning and end of each name.
- split() divides a name into separate words (e.g., first and last names).
- capitalize() ensures each part of the name starts with a **capital letter**.
- join() combines the cleaned words back into a single string.
- The result converts irregularly formatted names like " aISha khan " or "ALI Raza" into proper title case, e.g., "Aisha Khan", "Ali Raza", "Sara".

B9) Mini Grading Report (Dictionary + Loops + Formatting)

This task shows how to use **dictionaries**, **loops**, and **string formatting** to summarize students' marks and calculate averages.

```python
#B9) Mini grading report (dict + loops + formatting)
grades = {"Aisha":[78,82,91], "Ali":[65,70,68], "Sara":[90,95,87]}
for name, marks in grades.items():
    avg = sum(marks)/len(marks)
    print(f"{name:>5}: avg={avg:5.1f}, top={max(marks)}")
```

```
Aisha: avg= 83.7, top=91
  Ali: avg= 67.7, top=70
 Sara: avg= 90.7, top=95
```

Explanation:

- A dictionary grades stores student names as **keys** and their marks as **lists**.
- The for name, marks in grades.items() loop iterates through each student.
- sum(marks)/len(marks) calculates the **average score**.
- max(marks) finds the **highest mark** for each student.
- The formatted string f"{name:>5}: avg={avg:5.1f}, top={max(marks)}" neatly aligns names and numeric results for a clear, report-like output.

B10) CSV Read/Write (No Pandas)

This task demonstrates how to perform **file input/output operations** using Python's built-in **csv module** without relying on external libraries like Pandas.

```python
#B10) CSV read/write (no pandas)
import csv
rows = [["name","score"],["Aisha",88],["Ali",73]]
with open("scores.csv","w",newline="",encoding="utf-8") as f:
    csv.writer(f).writerows(rows)

with open("scores.csv","r",encoding="utf-8") as f:
    for row in csv.reader(f):
        print(row)
```

```
['name', 'score']
['Aisha', '88']
['Ali', '73']
```

Explanation:

- The csv module is used to handle **Comma-Separated Values** files.

- A list of lists rows stores tabular data with headers.
- csv.writer(f).writerows(rows) writes all rows into the file scores.csv.
- The second with open(...) block reads the file using csv.reader() and prints each row to verify the contents.
- Using the with statement ensures files are **automatically closed** after use.

Task 1 — Counting Odd and Even Numbers

This task counts how many numbers in a list are **odd** and how many are **even**, using **generator expressions** and the sum() function.

C1) Odd–Even Counter

```
# Task 1: Input: a list of integers. Output: count of odd and even numbers (print nicely).
def count_odd_even(nums):
    """Count how many numbers are odd or even."""
    odd = sum(1 for n in nums if n % 2 != 0)  # count odds
    even = sum(1 for n in nums if n % 2 == 0)  # count evens
    print(f"Odd: {odd}, Even: {even}")

assert count_odd_even([1, 2, 3, 4, 5, 6]) is None

# Explanation:
# Generator expressions count odd/even efficiently and print totals neatly.
```

```
Odd: 3, Even: 3
```

Explanation:

- The function iterates over all elements in the list nums.
- sum(1 for n in nums if n % 2 != 0) counts how many are **odd**.
- sum(1 for n in nums if n % 2 == 0) counts how many are **even**.
- The results are displayed neatly using **formatted strings (f-strings)**.
- The assert statement confirms that the function executes correctly and returns None since it only prints results.

Task 2 — String Reversal and Palindrome Check

This task demonstrates how to **reverse a string** and check if a string is a **palindrome**, ignoring spaces, punctuation, and case.

C2) String Utilities

```python
# Task 2: Write functions: reverse_string(s), is_palindrome(s) (ignore case and spaces).
def reverse_string(s):
    """Return the reversed version of a string."""
    return s[::-1]  # slicing reverses string

def is_palindrome(s):
    """Check if string is palindrome (ignore case/spaces)."""
    cleaned = ''.join(ch.lower() for ch in s if ch.isalnum())  # normalize
    return cleaned == cleaned[::-1]  # compare forward/backward

print("Reversed string:", reverse_string("hello"))
print("Is palindrome?:", is_palindrome("A man a plan a canal Panama"))

# Assertions (silent test checks)
assert reverse_string("hello") == "olleh"
assert is_palindrome("A man a plan a canal Panama")

# Explanation:
# Slicing reverses strings; palindrome check removes spaces and compares clean text both ways.
```

```
Reversed string: olleh
Is palindrome?: True
```

Explanation:

- The reverse_string() function uses **Python slicing [::-1]** to reverse a string.
- The is_palindrome() function:
    - Removes all **non-alphanumeric characters** and converts to lowercase for a clean comparison.
    - Compares the cleaned string with its reversed version to decide if it's a palindrome.
- The assert statements silently test both functions for correctness.

C3) List Flatten (Recursive)

```python
# Task 3: Given a nested list like [[1,2],[3,4],[5,[6]]], produce a flat list [1,2,3,4,5,6] (recursive).
def flatten_list(lst):
    """Flatten a nested list recursively."""
    flat = []
    for x in lst:
        if isinstance(x, list):          # if item is list, flatten it
            flat.extend(flatten_list(x)) # extend with inner items
        else:
            flat.append(x)               # else, append directly
    return flat

result = flatten_list([[1, 2], [3, 4], [5, [6]]])
print("Flattened list:", result)
assert result == [1, 2, 3, 4, 5, 6]

# Explanation:
# Uses recursion to unpack lists of any depth into a single flat list.
```

```
Flattened list: [1, 2, 3, 4, 5, 6]
```

C4) Top-k frequent words

```python
# Task 4: Given text (string), return the top 3 most frequent words with counts (use re and a dict).
import re
from collections import Counter

def top_k_words(text, k=3):
    """Return top-k frequent words with counts."""
    words = re.findall(r'\b\w+\b', text.lower())  # extract words
    counts = Counter(words)                       # count frequencies
    top_k = counts.most_common(k)                 # top-k list of tuples
    print("Top", k, "words with counts:", top_k)  # display results
    return top_k                                  # still return for testing

top_k_words("This is a test. This test is simple. Simple test!")
```

```
Top 3 words with counts: [('test', 3), ('this', 2), ('is', 2)]

[('test', 3), ('this', 2), ('is', 2)]
```

C5) Matrix Border Sum

```python
# Task 5: For a 2D list of numbers, compute the sum of the border elements (first/last row and column) without double counting corners.
def border_sum(matrix):
    """Sum all border elements (no double count)."""
    if not matrix or not matrix[0]:
        return 0  # handle empty matrix safely
    rows, cols = len(matrix), len(matrix[0])
    total = 0
    for i in range(rows):
        for j in range(cols):
            if i in [0, rows - 1] or j in [0, cols - 1]:  # check border
                total += matrix[i][j]
    return total

# Test + visible output
matrix = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]

result = border_sum(matrix)
print("Border sum:", result)
assert result == 40  # silent check

# Explanation:
# Adds all elements from first/last rows or columns; avoids double counting corners.
```

C6) Grade Bucketizer

Generate + Code + Markdown

```python
# Task 6: Convert numeric marks (0–100) to letter grades with thresholds and produce a frequency table (dict) of letter grades.
def grade_bucket(marks):
    """Convert marks into letter grades and count frequency."""

    def letter(m):  # helper to map marks to letter
        if m >= 90: return 'A'
        elif m >= 80: return 'B'
        elif m >= 70: return 'C'
        elif m >= 60: return 'D'
        else: return 'F'

    # Convert each mark to a letter using list comprehension
    grades = [letter(m) for m in marks]

    # Count frequency of each letter using dict comprehension
    freq = {g: grades.count(g) for g in set(grades)}

    return freq

marks = [95, 82, 76, 45, 89, 90, 60]
result = grade_bucket(marks)
print("Grade frequency:", result)
assert result["A"] == 2  # silent test check
```

C7) Unique Email Normalizer

Generate + Code + Markdown

```python
# Task 7: For a list of emails, treat “dots” ignored in local part and +tag suffixes as the same (Gmail-style). Count unique addresses.
def unique_emails(emails):
    """Count unique emails ignoring '.' and '+tag' parts."""

    # Use a set to store normalized emails (unique values only)
    normalized = {
        e.split('@')[0].split('+')[0].replace('.', '') + '@' + e.split('@')[1]
        for e in emails
    }

    return len(normalized)

emails = [
    "test.email+alex@gmail.com",
    "test.e.mail@gmail.com",
    "testemail@gmail.com"
]
result = unique_emails(emails)
print("Unique email count:", result)
assert result == 1  # silent correctness check

# Explanation:
# - For each email: remove dots ('.') and ignore anything after '+' before '@'.
# - The domain part (after '@') remains unchanged.
# - A set automatically removes duplicates.
```

C8) Inventory Merge

```python
# Task 8: You get two dicts of {item: qty}. Merge them by summing quantities; print items sorted by qty desc (ties by name asc).
def merge_inventories(inv1, inv2):
    """Merge two inventories, summing quantities and sorting results."""

    # Combine both inventories: sum quantities for shared items
    merged = {
        item: inv1.get(item, 0) + inv2.get(item, 0)
        for item in set(inv1) | set(inv2)  # union of both keys
    }

    # Sort items: highest quantity first, then alphabetically
    return sorted(merged.items(), key=lambda x: (-x[1], x[0]))

inv1 = {"apple": 5, "banana": 3}
inv2 = {"banana": 2, "orange": 7}

result = merge_inventories(inv1, inv2)
print("Merged inventory:", result)
assert result[0][0] == "orange"  # verify top item is 'orange'

# Explanation:
# - Uses dict comprehension with set union to merge keys.
# - Quantities summed using .get() for safe access.
# - Sorted by descending quantity and ascending name.
```

C9) Mini JSON Parser

```python
# Task 9: Input: a JSON string containing a list of people with name, age. Print the average age and the name(s) of the oldest.
import json

def people_stats(json_str):
    """Print average age and oldest person's name(s)."""
    try:
        # Parse JSON string into Python list of dicts
        people = json.loads(json_str)

        # Extract all ages
        ages = [p["age"] for p in people]

        # Compute average and max age
        avg = sum(ages) / len(ages)
        max_age = max(ages)

        # Get names of all people with max age
        oldest = [p["name"] for p in people if p["age"] == max_age]

        # Display formatted result
        print(f"Average age: {avg:.1f}")
        print(f"Oldest person(s): {', '.join(oldest)}")

    except Exception as e:
        # Catch and show errors for invalid JSON or data structure
        print("Error:", e)

sample = '[{"name": "Ali", "age": 20}, {"name": "Sara", "age": 25}, {"name": "Zain", "age": 25}]'
people_stats(sample)

# Explanation:
# - Loads JSON safely into Python objects.
# - Uses list comprehensions for average and max-age extraction.
# - Prints both average and oldest names clearly.
```

```
Average age: 23.3
Oldest person(s): Sara, Zain
```

File I/O Pipeline

```python
# Task 9: Read input.txt, remove lines that are empty or start with #, write the cleaned lines to output.txt. Show counts: total, kept, removed.
def clean_file(infile='input.txt', outfile='output.txt'):
    """Remove empty or # comment lines and save cleaned file."""
    total = kept = removed = 0
    try:
        # Open input file for reading and output file for writing
        with open(infile) as fin, open(outfile, 'w') as fout:
            for line in fin:
                total += 1
                text = line.strip()  # remove leading/trailing spaces

                # Skip blank or commented lines
                if not text or text.startswith('#'):
                    removed += 1
                else:
                    fout.write(text + '\n')  # write valid line
                    kept += 1

        # Print summary statistics
        print(f"Total: {total}, Kept: {kept}, Removed: {removed}")

    except FileNotFoundError:
        print(f"File '{infile}' not found.")
```

```
    # Create a temporary input file for testing
    with open('input.txt', 'w') as f:
        f.write("""# This is a comment
    Line 1

    Line 2
    # Another comment
    Line 3
    """)

    # Run the cleaner
    clean_file()

    with open('output.txt') as f:
        print("\nCleaned file content:")
        print(f.read())

    # Explanation:
    # - Reads each line safely.
    # - Skips comment/empty lines.
    # - Writes valid ones to output.txt and prints summary stats.
```

```
Total: 6, Kept: 3, Removed: 3

Cleaned file content:
Line 1
```