NAME : V.Shiva krishna

HALL TICKET No : 2303A52151

BATCH-41

TASK 1:

QUESTION:

Task 1: Privacy and Data Security in AI-Generated Code

Scenario

AI tools can sometimes generate insecure authentication logic.

Task Description

Use an AI tool to generate a simple login system in Python.

Analyze the generated code to check:

- Whether credentials are hardcoded
- Whether passwords are stored or compared in plain text
- Whether insecure logic is used

Then, revise the code to improve security (e.g., avoid hardcoding, use input validation).

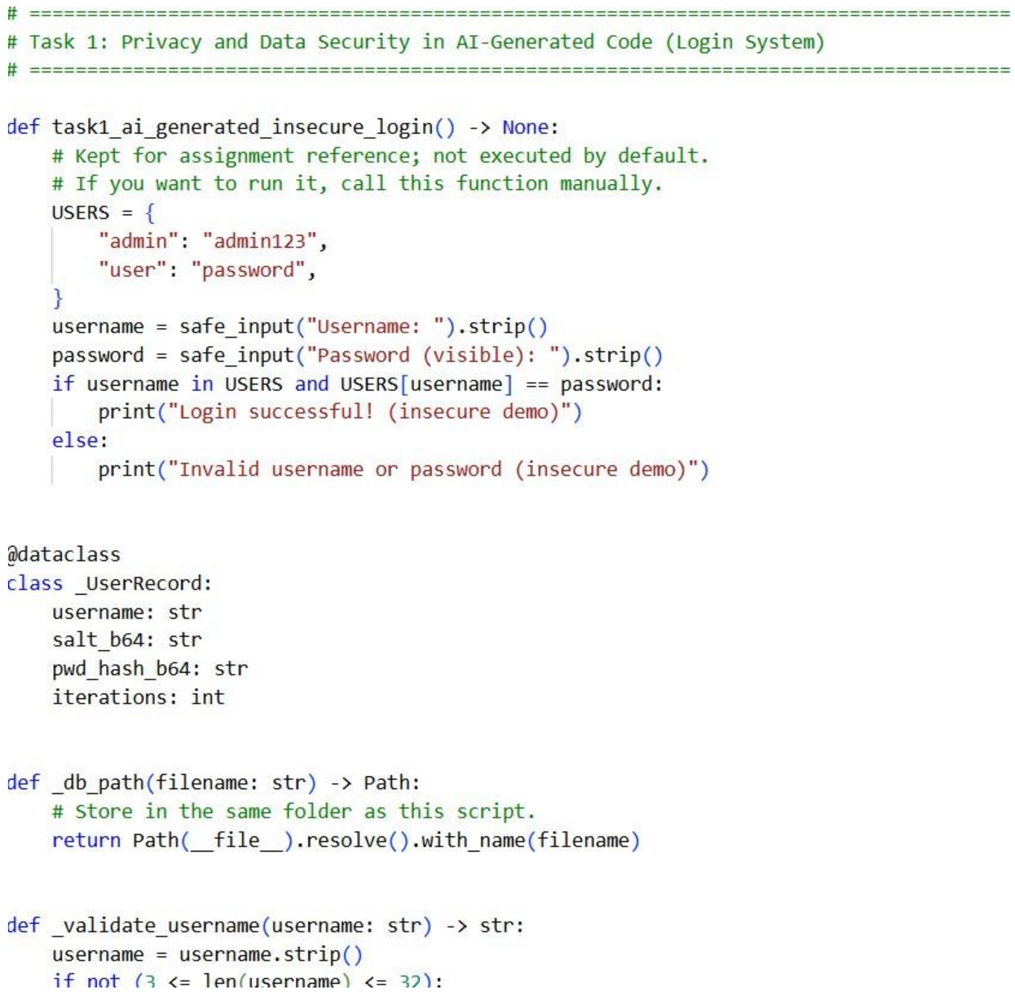
Expected Output

- AI-generated login code
- Identification of security risks
- Revised secure version of the code
- Brief explanation of improvements

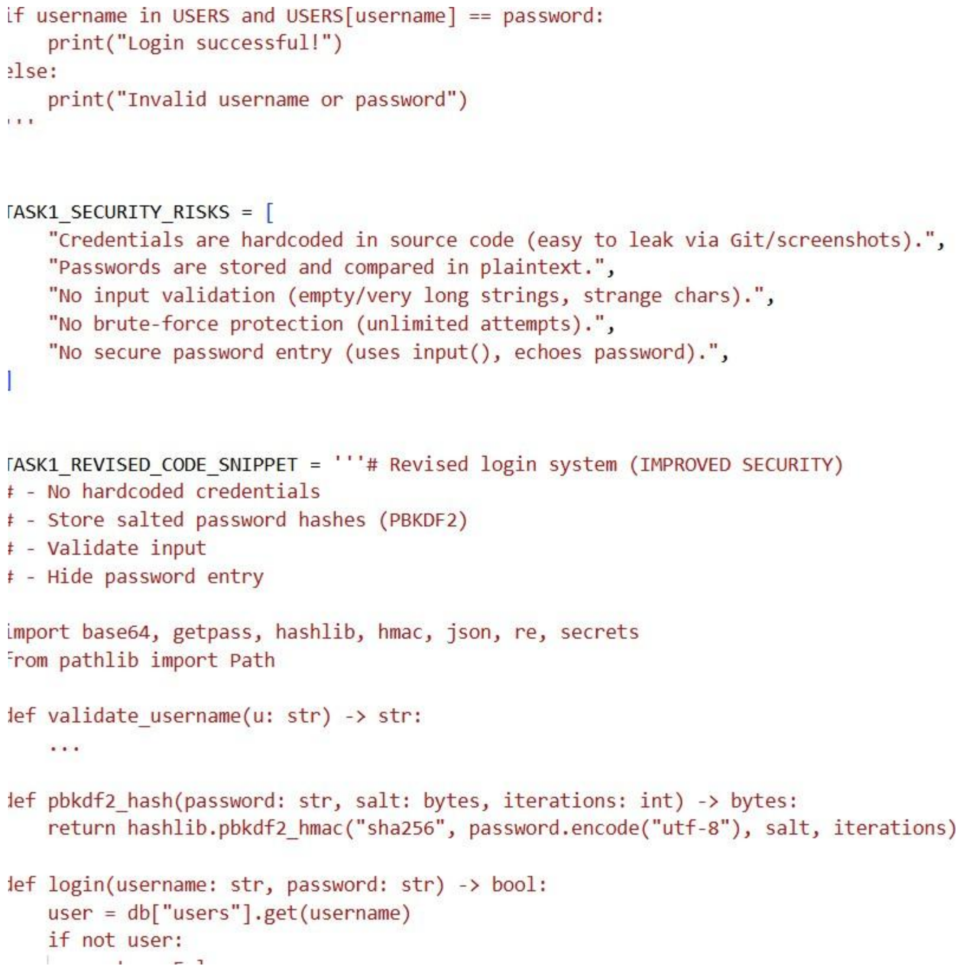
PROMPT:

Generate a simple login form which consists of the registration and login form to store in and inform it has login successful or unsuccessful. Do not create any external files and it consist of security risk and secure version of code in python only.

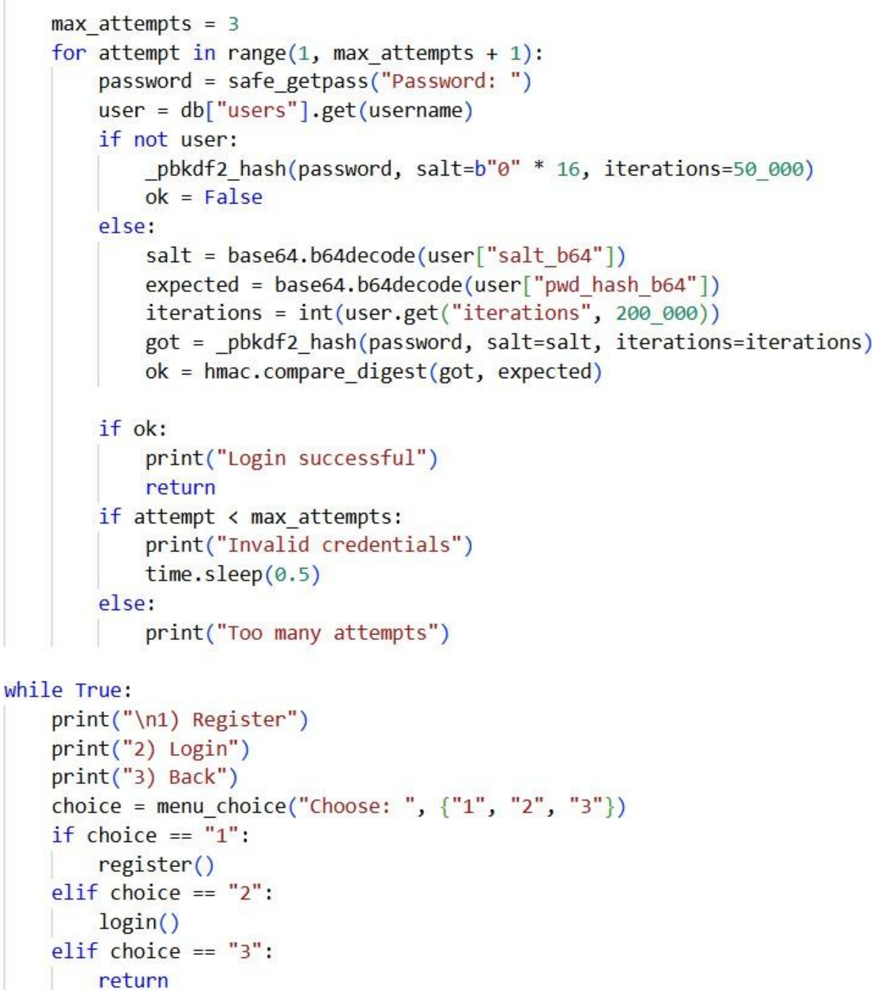
OUTPUT:



```
# ====================================================================================
# Task 1: Privacy and Data Security in AI-Generated Code (Login System)
# ====================================================================================

def task1_ai_generated_insecure_login() -> None:
    # Kept for assignment reference; not executed by default.
    # If you want to run it, call this function manually.
    USERS = {
        "admin": "admin123",
        "user": "password",
    }
    username = safe_input("Username: ").strip()
    password = safe_input("Password (visible): ").strip()
    if username in USERS and USERS[username] == password:
        print("Login successful! (insecure demo)")
    else:
        print("Invalid username or password (insecure demo)")


@dataclass
class _UserRecord:
    username: str
    salt_b64: str
    pwd_hash_b64: str
    iterations: int


def _db_path(filename: str) -> Path:
    # Store in the same folder as this script.
    return Path(__file__).resolve().with_name(filename)


def _validate_username(username: str) -> str:
    username = username.strip()
    if not (3 <= len(username) <= 32):
```

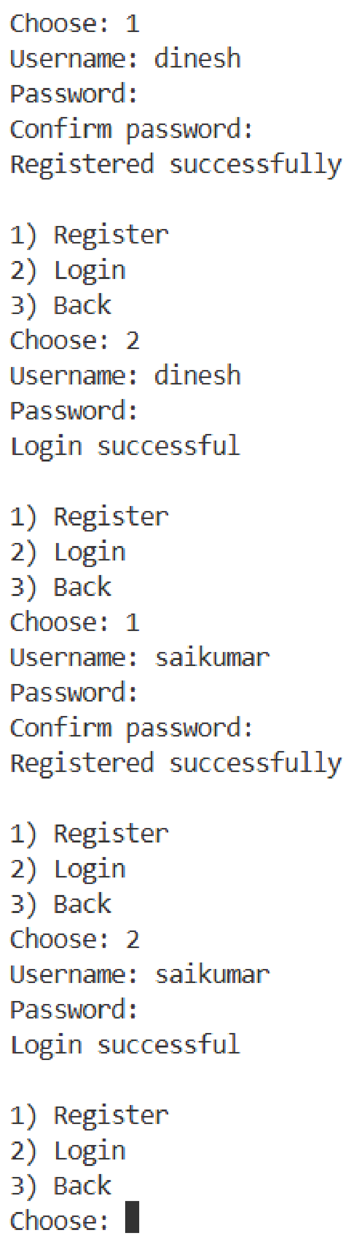


```
if username in USERS and USERS[username] == password:
    print("Login successful!")
else:
    print("Invalid username or password")
'''


TASK1_SECURITY_RISKS = [
    "Credentials are hardcoded in source code (easy to leak via Git/screenshots).",
    "Passwords are stored and compared in plaintext.",
    "No input validation (empty/very long strings, strange chars).",
    "No brute-force protection (unlimited attempts).",
    "No secure password entry (uses input(), echoes password).",
]


TASK1_REVISED_CODE_SNIPPET = '''# Revised login system (IMPROVED SECURITY)
# - No hardcoded credentials
# - Store salted password hashes (PBKDF2)
# - Validate input
# - Hide password entry

import base64, getpass, hashlib, hmac, json, re, secrets
from pathlib import Path

def validate_username(u: str) -> str:
    ...

def pbkdf2_hash(password: str, salt: bytes, iterations: int) -> bytes:
    return hashlib.pbkdf2_hmac("sha256", password.encode("utf-8"), salt, iterations)

def login(username: str, password: str) -> bool:
    user = db["users"].get(username)
    if not user:
```



```
    max_attempts = 3
    for attempt in range(1, max_attempts + 1):
        password = safe_getpass("Password: ")
        user = db["users"].get(username)
        if not user:
            _pbkdf2_hash(password, salt=b"0" * 16, iterations=50_000)
            ok = False
        else:
            salt = base64.b64decode(user["salt_b64"])
            expected = base64.b64decode(user["pwd_hash_b64"])
            iterations = int(user.get("iterations", 200_000))
            got = _pbkdf2_hash(password, salt=salt, iterations=iterations)
            ok = hmac.compare_digest(got, expected)

        if ok:
            print("Login successful")
            return
        if attempt < max_attempts:
            print("Invalid credentials")
            time.sleep(0.5)
        else:
            print("Too many attempts")

while True:
    print("\n1) Register")
    print("2) Login")
    print("3) Back")
    choice = menu_choice("Choose: ", {"1", "2", "3"})
    if choice == "1":
        register()
    elif choice == "2":
        login()
    elif choice == "3":
        return
```

OUTPUT:



```
Choose: 1
Username: dinesh
Password:
Confirm password:
Registered successfully

1) Register
2) Login
3) Back
Choose: 2
Username: dinesh
Password:
Login successful

1) Register
2) Login
3) Back
Choose: 1
Username: saikumar
Password:
Confirm password:
Registered successfully

1) Register
2) Login
3) Back
Choose: 2
Username: saikumar
Password:
Login successful

1) Register
2) Login
3) Back
Choose: █
```

JUSTIFICATION:

This prompt was chosen to intentionally generate basic AI-written login code, which often contains common security flaws. By analyzing and revising the code, the task demonstrates how AI-generated programs must be reviewed for vulnerabilities and improved using secure coding practices. It emphasizes responsible use of AI in software development and awareness of security risks.

Task 2:

Bias Detection in AI-Generated Decision Systems

Scenario

AI systems may unintentionally introduce bias.

Task Description

Use AI prompts such as:

- “Create a loan approval system”
- Vary applicant names and genders in prompts

Analyze whether:

- The logic treats certain genders or names unfairly
- Approval decisions depend on irrelevant personal attributes

Suggest methods to reduce or remove bias.

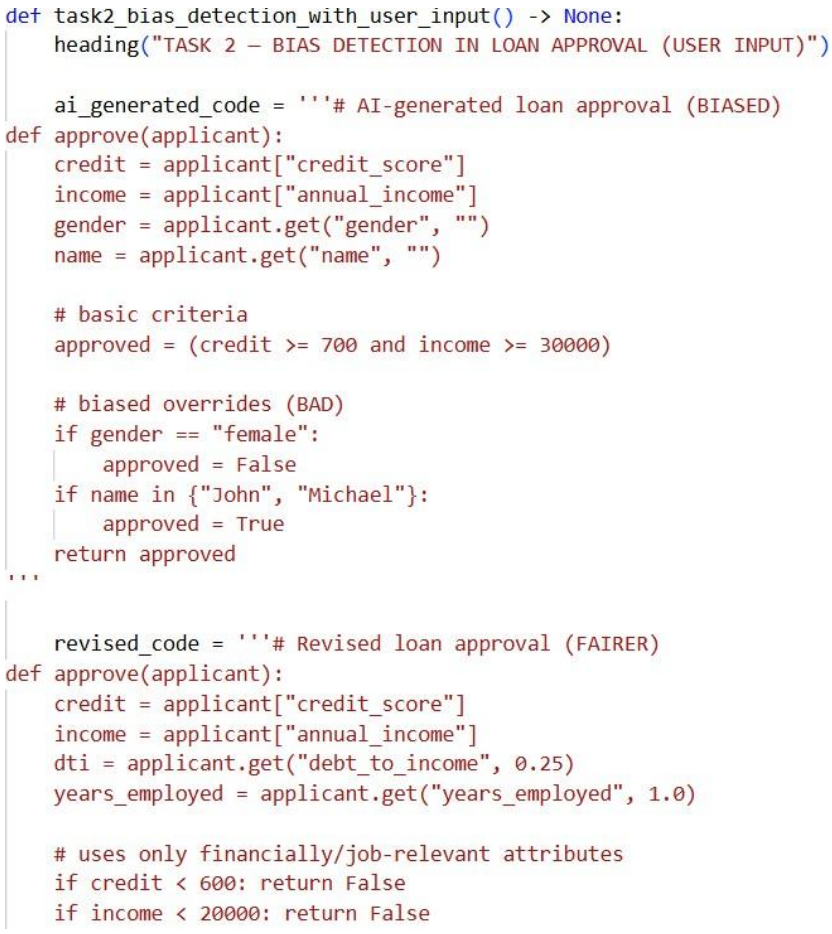
Expected Output

- Python code generated by AI
- Identification of biased logic (if any)
- Discussion on fairness issues
- Mitigation strategies

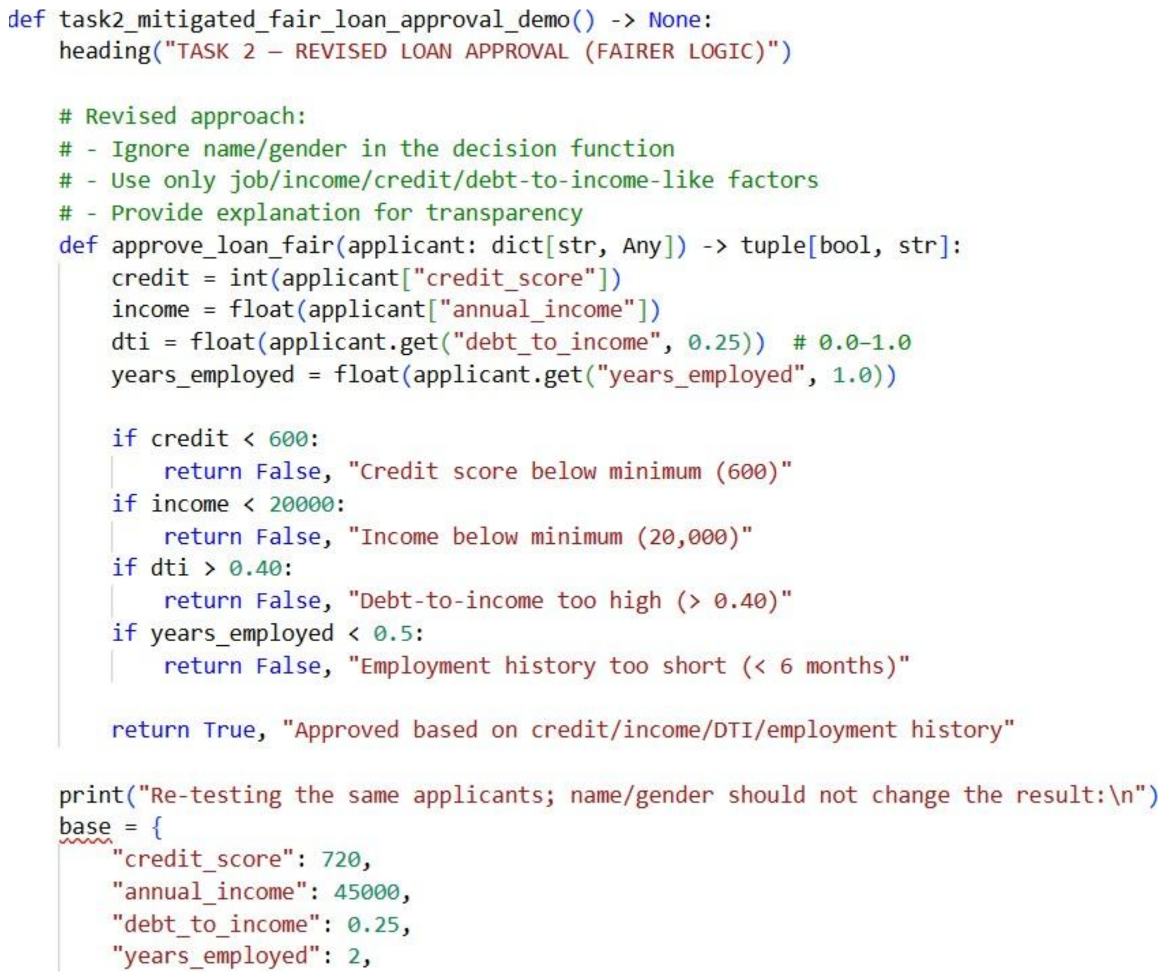
PROMPT:

Create a Python-based loan approval system. The prompt was varied by changing applicant names and genders while keeping financial details similar to observe differences in outcomes.

CODE:

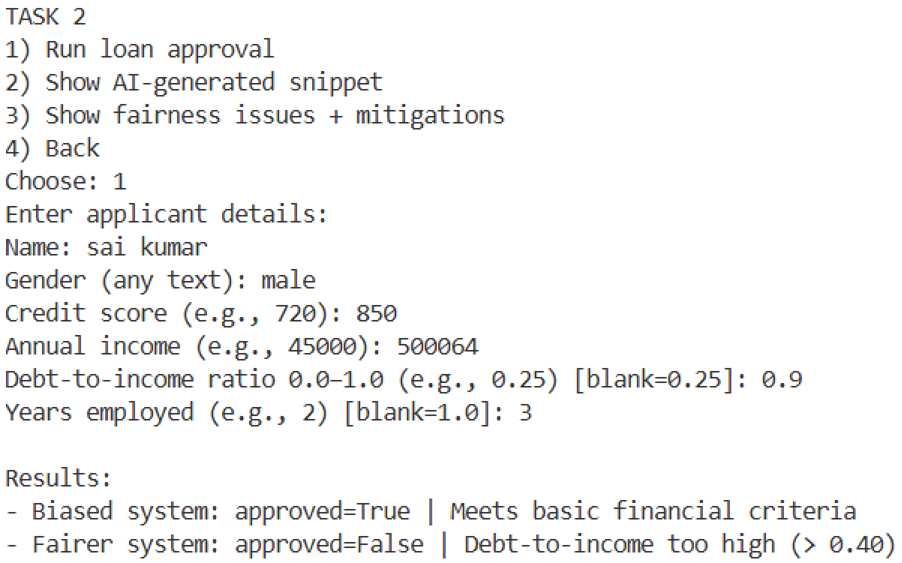


```
def task2_bias_detection_with_user_input() -> None:
    heading("TASK 2 — BIAS DETECTION IN LOAN APPROVAL (USER INPUT)")

    ai_generated_code = '''# AI-generated loan approval (BIASED)
def approve(applicant):
    credit = applicant["credit_score"]
    income = applicant["annual_income"]
    gender = applicant.get("gender", "")
    name = applicant.get("name", "")

    # basic criteria
    approved = (credit >= 700 and income >= 30000)

    # biased overrides (BAD)
    if gender == "female":
        approved = False
    if name in {"John", "Michael"}:
        approved = True
    return approved
'''

    revised_code = '''# Revised loan approval (FAIRER)
def approve(applicant):
    credit = applicant["credit_score"]
    income = applicant["annual_income"]
    dti = applicant.get("debt_to_income", 0.25)
    years_employed = applicant.get("years_employed", 1.0)

    # uses only financially/job-relevant attributes
    if credit < 600: return False
    if income < 20000: return False
```

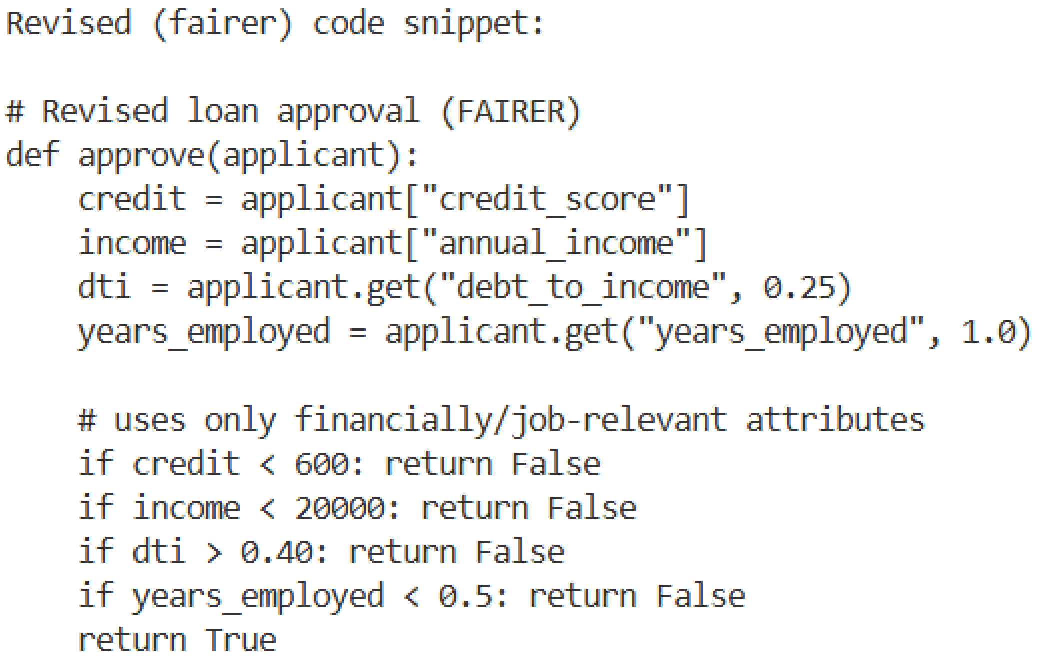


```
def task2_mitigated_fair_loan_approval_demo() -> None:
    heading("TASK 2 — REVISED LOAN APPROVAL (FAIRER LOGIC)")

    # Revised approach:
    # - Ignore name/gender in the decision function
    # - Use only job/income/credit/debt-to-income-like factors
    # - Provide explanation for transparency
    def approve_loan_fair(applicant: dict[str, Any]) -> tuple[bool, str]:
        credit = int(applicant["credit_score"])
        income = float(applicant["annual_income"])
        dti = float(applicant.get("debt_to_income", 0.25))  # 0.0-1.0
        years_employed = float(applicant.get("years_employed", 1.0))

        if credit < 600:
            return False, "Credit score below minimum (600)"
        if income < 20000:
            return False, "Income below minimum (20,000)"
        if dti > 0.40:
            return False, "Debt-to-income too high (> 0.40)"
        if years_employed < 0.5:
            return False, "Employment history too short (< 6 months)"

        return True, "Approved based on credit/income/DTI/employment history"

    print("Re-testing the same applicants; name/gender should not change the result:\n")
    base = {
        "credit_score": 720,
        "annual_income": 45000,
        "debt_to_income": 0.25,
        "years_employed": 2,
```

OUTPUT:



```
TASK 2
1) Run loan approval
2) Show AI-generated snippet
3) Show fairness issues + mitigations
4) Back
Choose: 1
Enter applicant details:
Name: sai kumar
Gender (any text): male
Credit score (e.g., 720): 850
Annual income (e.g., 45000): 500064
Debt-to-income ratio 0.0-1.0 (e.g., 0.25) [blank=0.25]: 0.9
Years employed (e.g., 2) [blank=1.0]: 3

Results:
- Biased system: approved=True | Meets basic financial criteria
- Fairer system: approved=False | Debt-to-income too high (> 0.40)
```



Revised (fairer) code snippet:

```
# Revised loan approval (FAIRER)
def approve(applicant):
    credit = applicant["credit_score"]
    income = applicant["annual_income"]
    dti = applicant.get("debt_to_income", 0.25)
    years_employed = applicant.get("years_employed", 1.0)

    # uses only financially/job-relevant attributes
    if credit < 600: return False
    if income < 20000: return False
    if dti > 0.40: return False
    if years_employed < 0.5: return False
    return True
```

JUSTIFICATION:

Loan approval systems have significant real-world impact, and this task helps identify whether AI logic unfairly treats individuals based on gender or name. Varying personal attributes allows analysis of bias in decision-making and highlights the importance of fairness and ethical considerations in AI-generated systems.

Task 3:

Transparency and Explainability in AI-Generated Code (Recursive

Binary Search)

Scenario

AI-generated code should be transparent, well-documented, and easy for

humans to understand and verify.

Task Description

Use an AI tool to generate a Python program that:

- Implements Binary Search using recursion
- Searches for a given element in a sorted list
- Includes:
    - o Clear inline comments
    - o A step-by-step explanation of the recursive logic

After generating the code, analyze:

- Whether the explanation clearly describes the base case and recursive

case

- Whether the comments correctly match the code logic
- Whether the code is understandable for beginner-level students

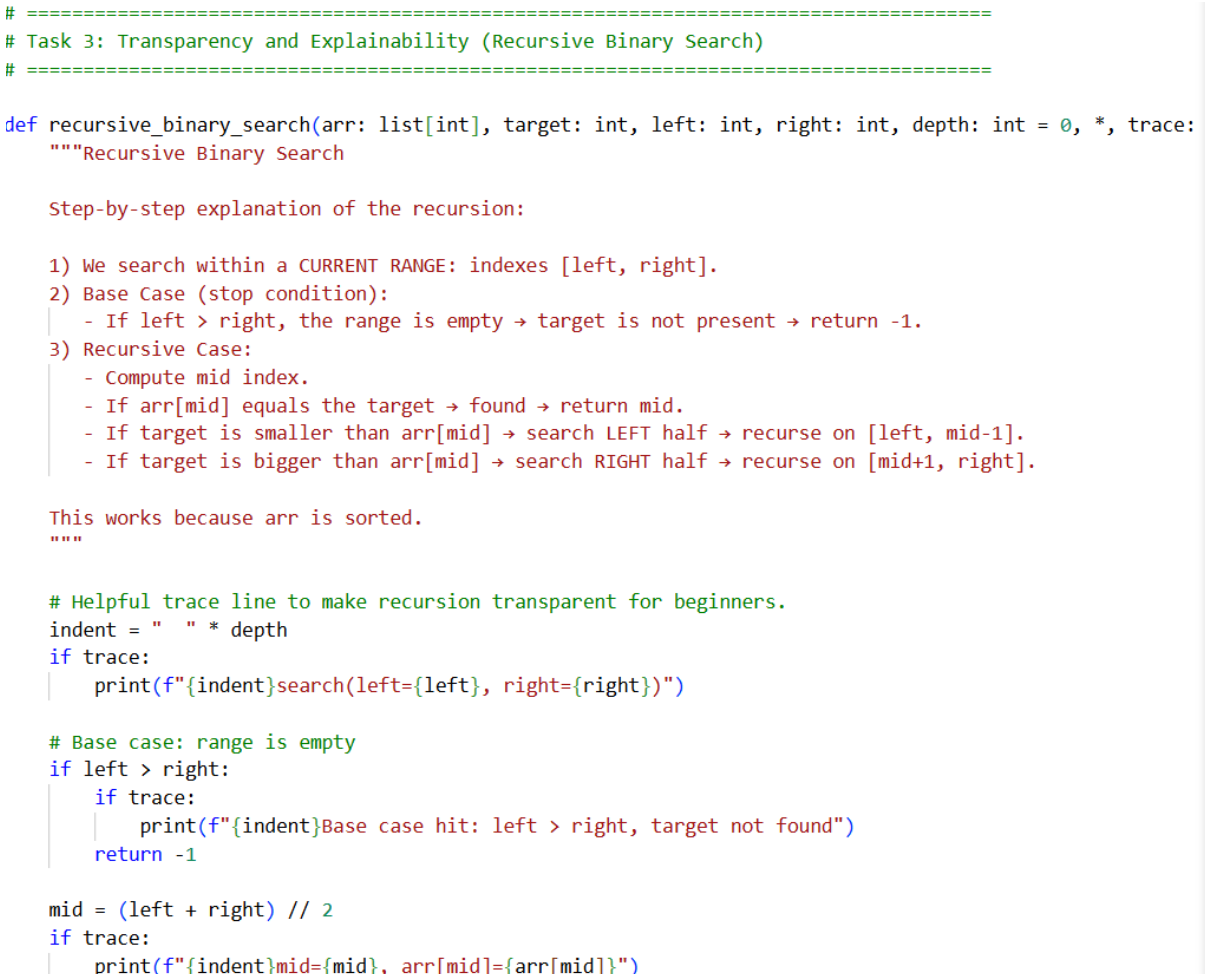
Expected Output

- Python program for recursive binary search
- AI-generated comments and explanation
- Student’s assessment on clarity, correctness, and transparency

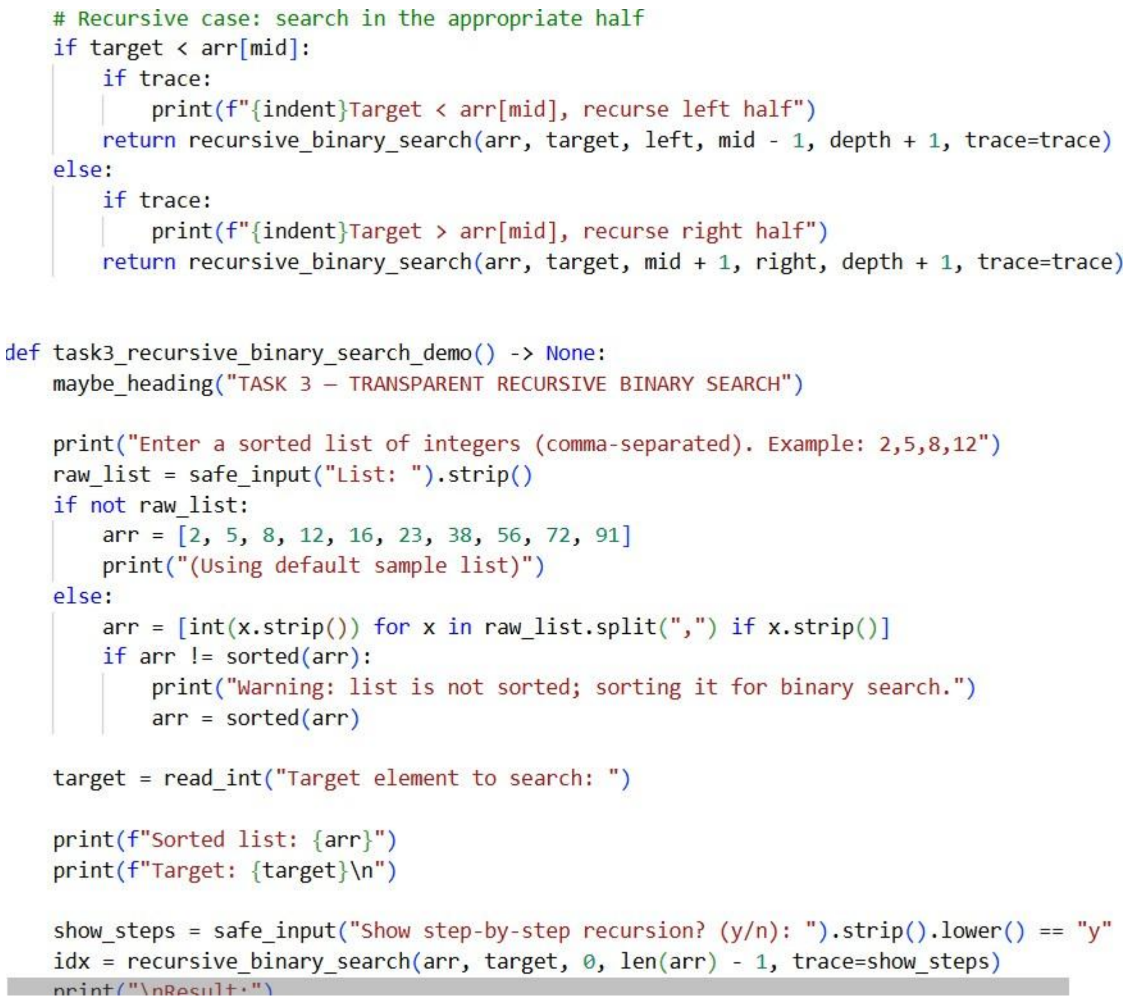
PROMPT:

Generate a Python program that implements binary search using recursion, including clear inline comments and a step-by-step explanation of the recursive logic.

CODE:

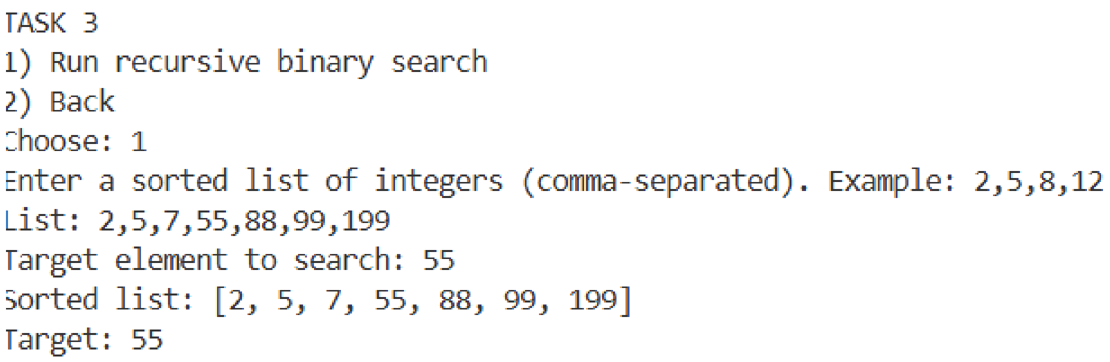


```
# ==================================================================================
# Task 3: Transparency and Explainability (Recursive Binary Search)
# ==================================================================================

def recursive_binary_search(arr: list[int], target: int, left: int, right: int, depth: int = 0, *, trace:
    """Recursive Binary Search

    Step-by-step explanation of the recursion:

    1) We search within a CURRENT RANGE: indexes [left, right].
    2) Base Case (stop condition):
       - If left > right, the range is empty → target is not present → return -1.
    3) Recursive Case:
       - Compute mid index.
       - If arr[mid] equals the target → found → return mid.
       - If target is smaller than arr[mid] → search LEFT half → recurse on [left, mid-1].
       - If target is bigger than arr[mid] → search RIGHT half → recurse on [mid+1, right].

    This works because arr is sorted.
    """

    # Helpful trace line to make recursion transparent for beginners.
    indent = "  " * depth
    if trace:
        print(f"{indent}search(left={left}, right={right})")

    # Base case: range is empty
    if left > right:
        if trace:
            print(f"{indent}Base case hit: left > right, target not found")
        return -1

    mid = (left + right) // 2
    if trace:
        print(f"{indent}mid={mid}, arr[mid]={arr[mid]}")
```

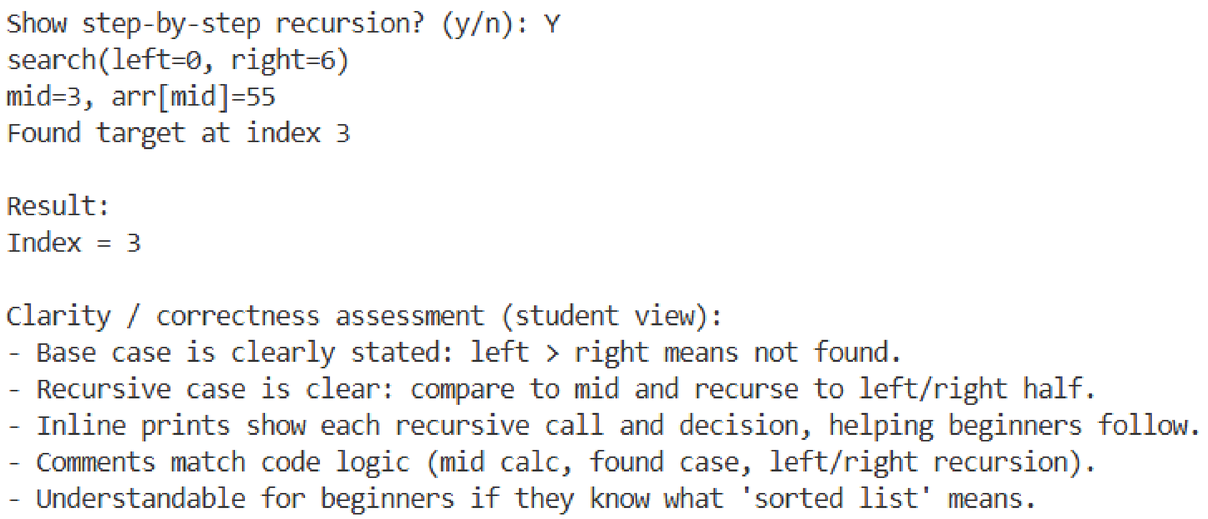


```
    # Recursive case: search in the appropriate half
    if target < arr[mid]:
        if trace:
            print(f"{indent}Target < arr[mid], recurse left half")
        return recursive_binary_search(arr, target, left, mid - 1, depth + 1, trace=trace)
    else:
        if trace:
            print(f"{indent}Target > arr[mid], recurse right half")
        return recursive_binary_search(arr, target, mid + 1, right, depth + 1, trace=trace)


def task3_recursive_binary_search_demo() -> None:
    maybe_heading("TASK 3 — TRANSPARENT RECURSIVE BINARY SEARCH")

    print("Enter a sorted list of integers (comma-separated). Example: 2,5,8,12")
    raw_list = safe_input("List: ").strip()
    if not raw_list:
        arr = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
        print("(Using default sample list)")
    else:
        arr = [int(x.strip()) for x in raw_list.split(",") if x.strip()]
        if arr != sorted(arr):
            print("Warning: list is not sorted; sorting it for binary search.")
            arr = sorted(arr)

    target = read_int("Target element to search: ")

    print(f"Sorted list: {arr}")
    print(f"Target: {target}\n")

    show_steps = safe_input("Show step-by-step recursion? (y/n): ").strip().lower() == "y"
    idx = recursive_binary_search(arr, target, 0, len(arr) - 1, trace=show_steps)
    print("\nResult:")
```

OUTPUT:



```
TASK 3
1) Run recursive binary search
2) Back
Choose: 1
Enter a sorted list of integers (comma-separated). Example: 2,5,8,12
List: 2,5,7,55,88,99,199
Target element to search: 55
Sorted list: [2, 5, 7, 55, 88, 99, 199]
Target: 55
```



```
Show step-by-step recursion? (y/n): Y
search(left=0, right=6)
mid=3, arr[mid]=55
Found target at index 3

Result:
Index = 3

Clarity / correctness assessment (student view):
- Base case is clearly stated: left > right means not found.
- Recursive case is clear: compare to mid and recurse to left/right half.
- Inline prints show each recursive call and decision, helping beginners follow.
- Comments match code logic (mid calc, found case, left/right recursion).
- Understandable for beginners if they know what 'sorted list' means.
```

JUSTIFICATION;

This prompt ensures that the AI not only generates functional code but also provides explanations that make the logic transparent. It helps evaluate whether AI-generated explanations clearly describe base cases and recursive calls and whether the code is understandable for beginner-level students, promoting explainable AI.

Task 4:

Ethical Evaluation of AI-Based Scoring Systems

Scenario

AI-generated scoring systems can influence hiring decisions.

Task Description

Ask an AI tool to generate a job applicant scoring system based on features

such as:

- Skills
- Experience
- Education

Analyze the generated code to check:

- Whether gender, name, or unrelated features influence scoring
- Whether the logic is fair and objective

Expected Output

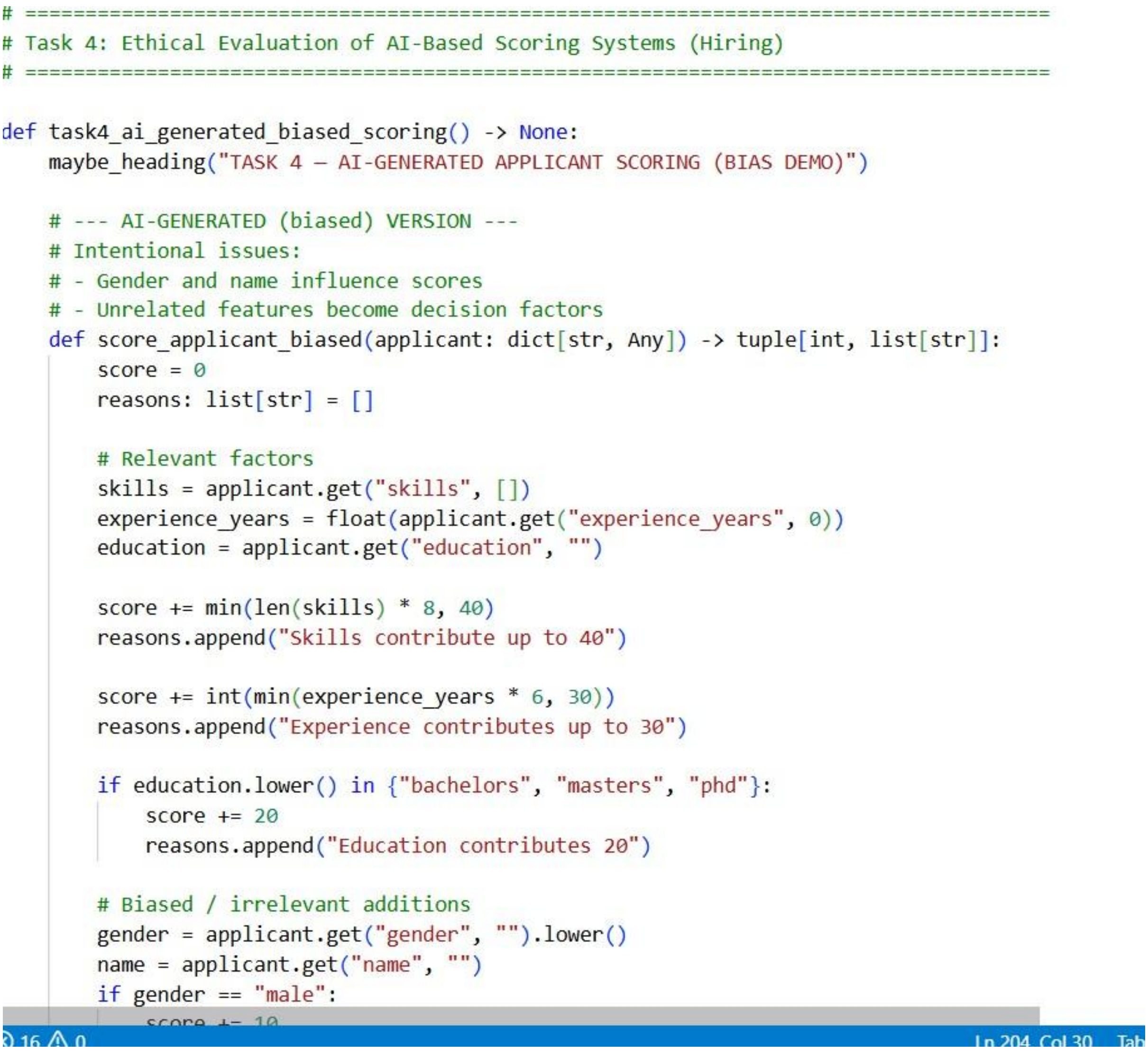
- Python scoring system code

- Identification of potential bias (if any)
- Ethical analysis of the scoring logic

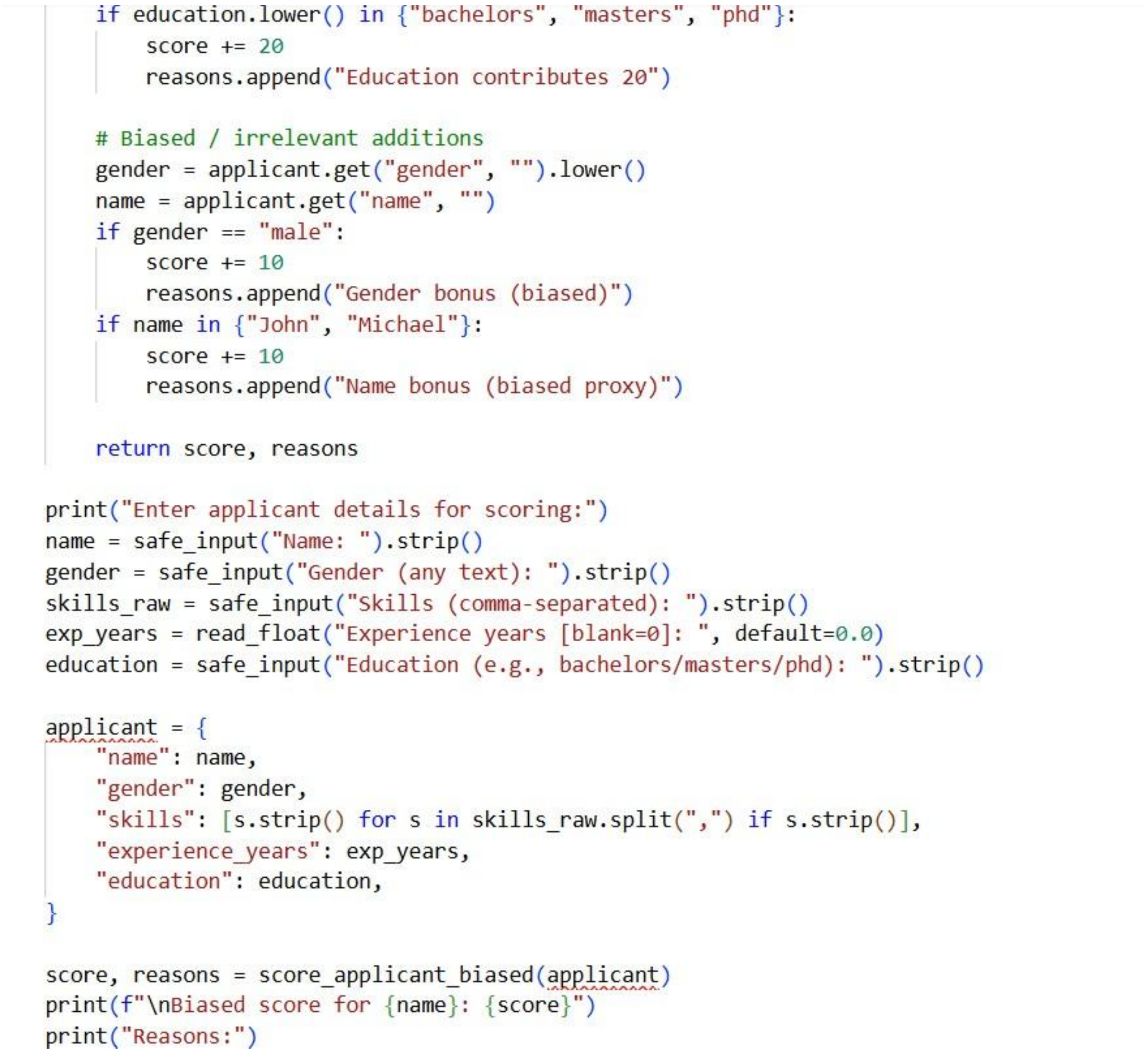
PROMPT:

Generate a Python job applicant scoring system based on skills, experience, and education.

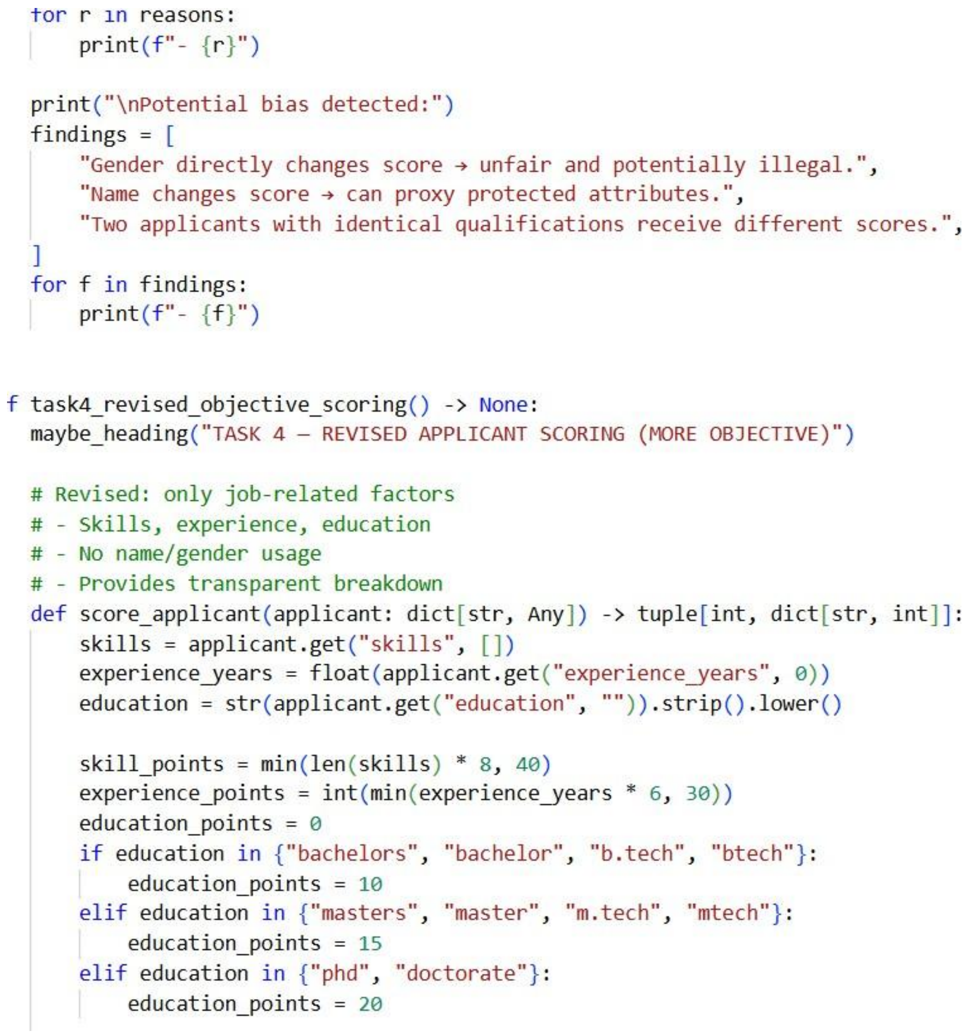
CODE:



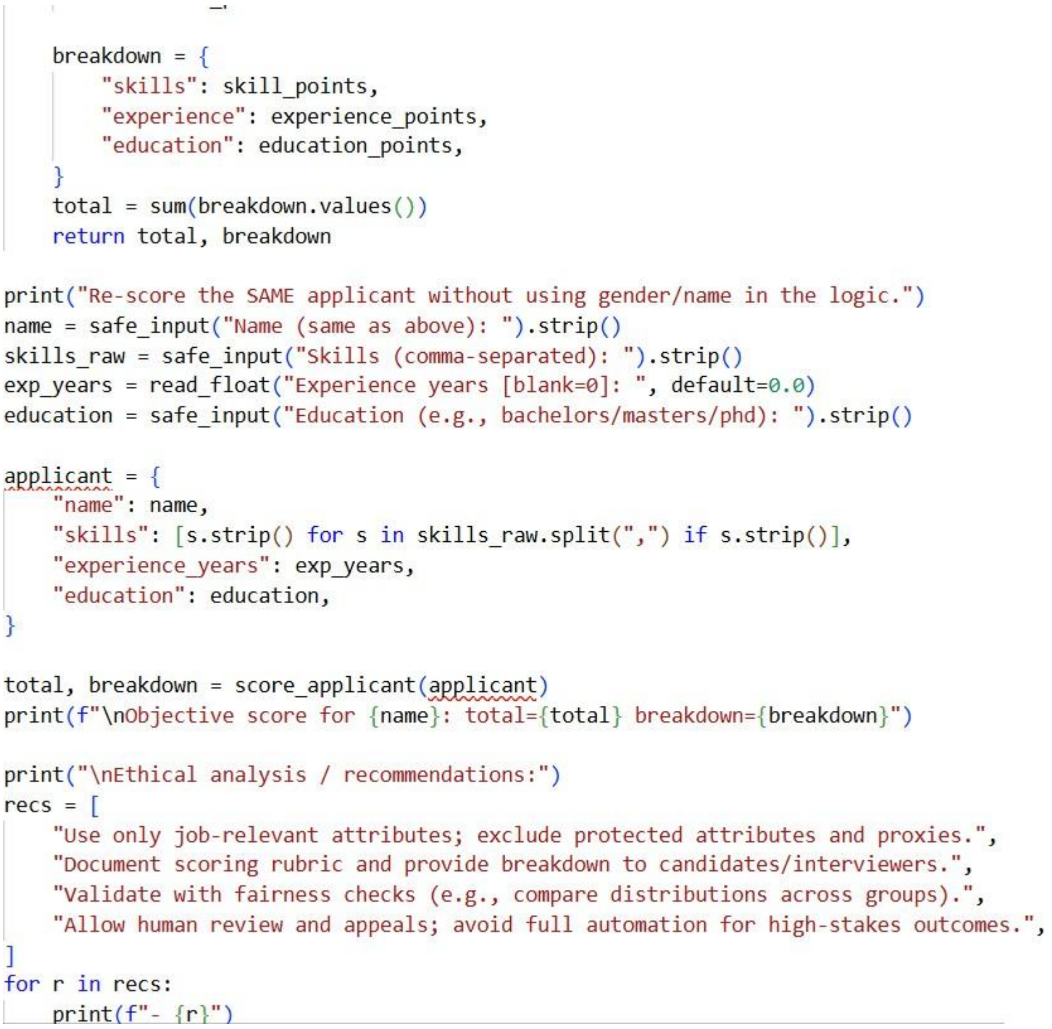
```
# ====================================================================================
# Task 4: Ethical Evaluation of AI-Based Scoring Systems (Hiring)
# ====================================================================================

def task4_ai_generated_biased_scoring() -> None:
    maybe_heading("TASK 4 — AI-GENERATED APPLICANT SCORING (BIAS DEMO)")

    # --- AI-GENERATED (biased) VERSION ---
    # Intentional issues:
    # - Gender and name influence scores
    # - Unrelated features become decision factors
    def score_applicant_biased(applicant: dict[str, Any]) -> tuple[int, list[str]]:
        score = 0
        reasons: list[str] = []

        # Relevant factors
        skills = applicant.get("skills", [])
        experience_years = float(applicant.get("experience_years", 0))
        education = applicant.get("education", "")

        score += min(len(skills) * 8, 40)
        reasons.append("Skills contribute up to 40")

        score += int(min(experience_years * 6, 30))
        reasons.append("Experience contributes up to 30")

        if education.lower() in {"bachelors", "masters", "phd"}:
            score += 20
            reasons.append("Education contributes 20")

        # Biased / irrelevant additions
        gender = applicant.get("gender", "").lower()
        name = applicant.get("name", "")
        if gender == "male":
            score += 10
```



```python
    if education.lower() in {"bachelors", "masters", "phd"}:
        score += 20
        reasons.append("Education contributes 20")

    # Biased / irrelevant additions
    gender = applicant.get("gender", "").lower()
    name = applicant.get("name", "")
    if gender == "male":
        score += 10
        reasons.append("Gender bonus (biased)")
    if name in {"John", "Michael"}:
        score += 10
        reasons.append("Name bonus (biased proxy)")

    return score, reasons

print("Enter applicant details for scoring:")
name = safe_input("Name: ").strip()
gender = safe_input("Gender (any text): ").strip()
skills_raw = safe_input("Skills (comma-separated): ").strip()
exp_years = read_float("Experience years [blank=0]: ", default=0.0)
education = safe_input("Education (e.g., bachelors/masters/phd): ").strip()

applicant = {
    "name": name,
    "gender": gender,
    "skills": [s.strip() for s in skills_raw.split(",") if s.strip()],
    "experience_years": exp_years,
    "education": education,
}

score, reasons = score_applicant_biased(applicant)
print(f"\nBiased score for {name}: {score}")
print("Reasons:")
```

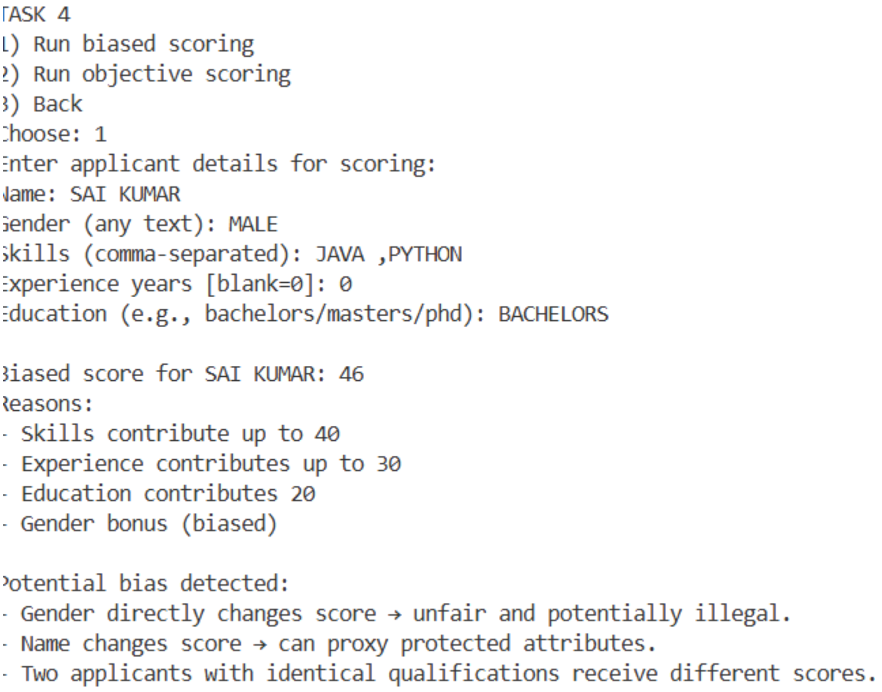


```
    for r in reasons:
        print(f"- {r}")

    print("\nPotential bias detected:")
    findings = [
        "Gender directly changes score → unfair and potentially illegal.",
        "Name changes score → can proxy protected attributes.",
        "Two applicants with identical qualifications receive different scores.",
    ]
    for f in findings:
        print(f"- {f}")


  f task4_revised_objective_scoring() -> None:
    maybe_heading("TASK 4 — REVISED APPLICANT SCORING (MORE OBJECTIVE)")

    # Revised: only job-related factors
    # - Skills, experience, education
    # - No name/gender usage
    # - Provides transparent breakdown
    def score_applicant(applicant: dict[str, Any]) -> tuple[int, dict[str, int]]:
        skills = applicant.get("skills", [])
        experience_years = float(applicant.get("experience_years", 0))
        education = str(applicant.get("education", "")).strip().lower()

        skill_points = min(len(skills) * 8, 40)
        experience_points = int(min(experience_years * 6, 30))
        education_points = 0
        if education in {"bachelors", "bachelor", "b.tech", "btech"}:
            education_points = 10
        elif education in {"masters", "master", "m.tech", "mtech"}:
            education_points = 15
        elif education in {"phd", "doctorate"}:
            education_points = 20
```



```
    breakdown = {
        "skills": skill_points,
        "experience": experience_points,
        "education": education_points,
    }
    total = sum(breakdown.values())
    return total, breakdown

print("Re-score the SAME applicant without using gender/name in the logic.")
name = safe_input("Name (same as above): ").strip()
skills_raw = safe_input("Skills (comma-separated): ").strip()
exp_years = read_float("Experience years [blank=0]: ", default=0.0)
education = safe_input("Education (e.g., bachelors/masters/phd): ").strip()

applicant = {
    "name": name,
    "skills": [s.strip() for s in skills_raw.split(",") if s.strip()],
    "experience_years": exp_years,
    "education": education,
}

total, breakdown = score_applicant(applicant)
print(f"\nObjective score for {name}: total={total} breakdown={breakdown}")

print("\nEthical analysis / recommendations:")
recs = [
    "Use only job-relevant attributes; exclude protected attributes and proxies.",
    "Document scoring rubric and provide breakdown to candidates/interviewers.",
    "Validate with fairness checks (e.g., compare distributions across groups).",
    "Allow human review and appeals; avoid full automation for high-stakes outcomes.",
]
for r in recs:
    print(f"- {r}")
```

OUTPUT:



```
ASK 4
) Run biased scoring
) Run objective scoring
) Back
hoose: 1
nter applicant details for scoring:
ame: SAI KUMAR
ender (any text): MALE
kills (comma-separated): JAVA ,PYTHON
xperience years [blank=0]: 0
ducation (e.g., bachelors/masters/phd): BACHELORS

iased score for SAI KUMAR: 46
easons:
 Skills contribute up to 40
 Experience contributes up to 30
 Education contributes 20
 Gender bonus (biased)

otential bias detected:
 Gender directly changes score → unfair and potentially illegal.
 Name changes score → can proxy protected attributes.
 Two applicants with identical qualifications receive different scores.
```

JUSTIFICATION:

Hiring and scoring systems directly influence people's careers, making ethical evaluation essential. This task checks whether AI-generated logic includes bias or uses irrelevant personal attributes. It reinforces the need for objective, fair, and ethical AI systems in decision-making processes.

Task 5:

Inclusiveness and Ethical Variable Design

Scenario

Inclusive coding practices avoid assumptions related to gender, identity, or roles and promote fairness in software design.

Task Description

Use an AI tool to generate a Python code snippet that processes user or employee details.

Analyze the code to identify:

- Gender-specific variables (e.g., male, female)
- Assumptions based on gender or identity
- Non-inclusive naming or logic

Modify or regenerate the code to:

- Use gender-neutral variable names
- Avoid gender-based conditions unless strictly required
- Ensure inclusive and respectful coding practices

Expected Output

- Original AI-generated code snippet
- Revised inclusive and gender-neutral code
- Brief explanation of:

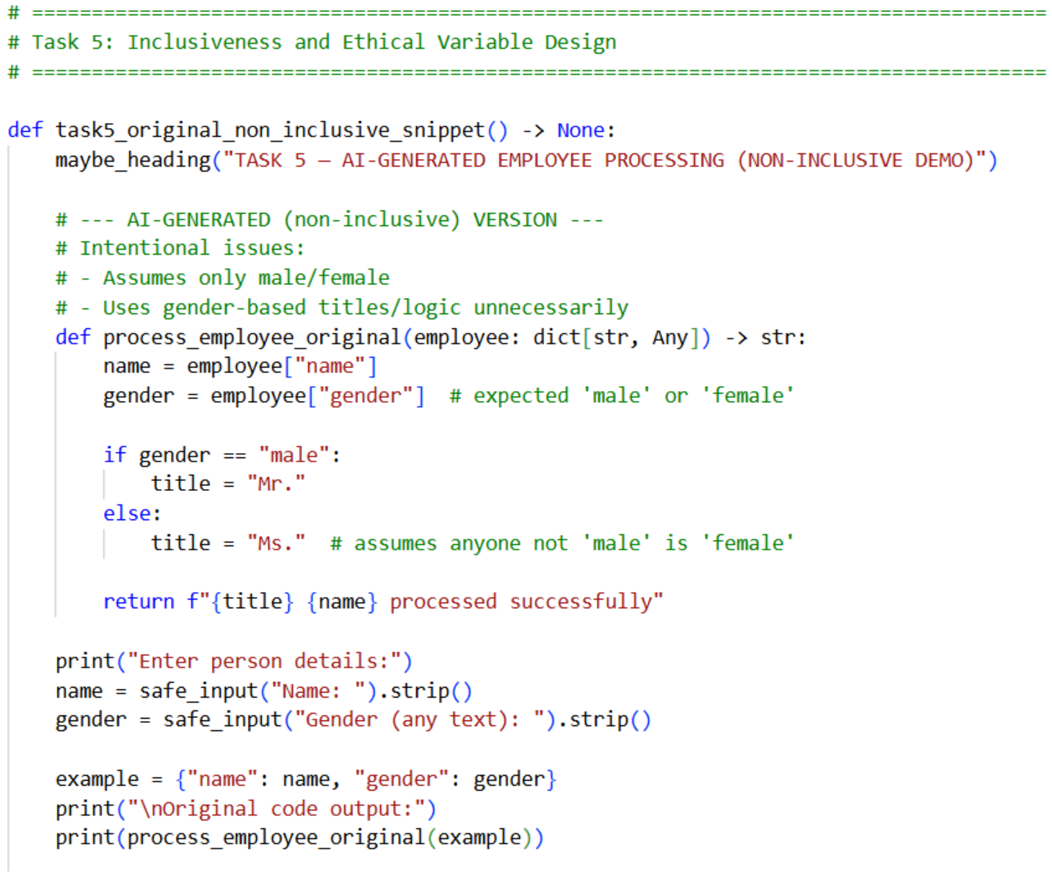
o What was non-inclusive

o How inclusiveness was improved

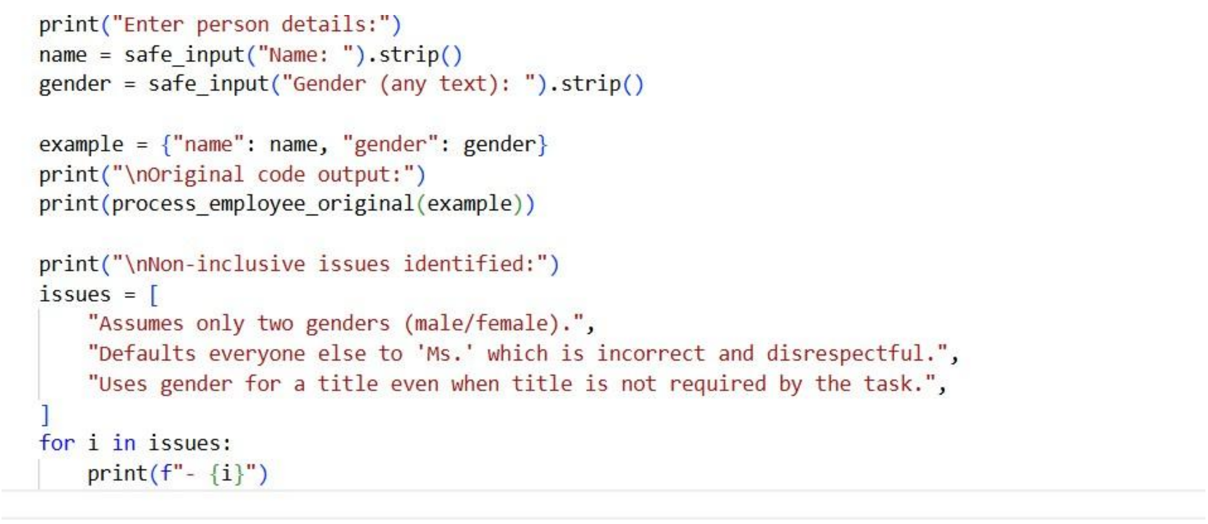
PROMPT:

Generate a Python program that processes employee details including name, age, gender, and role.A follow-up prompt was used to revise the code to be gender-neutral and inclusive.

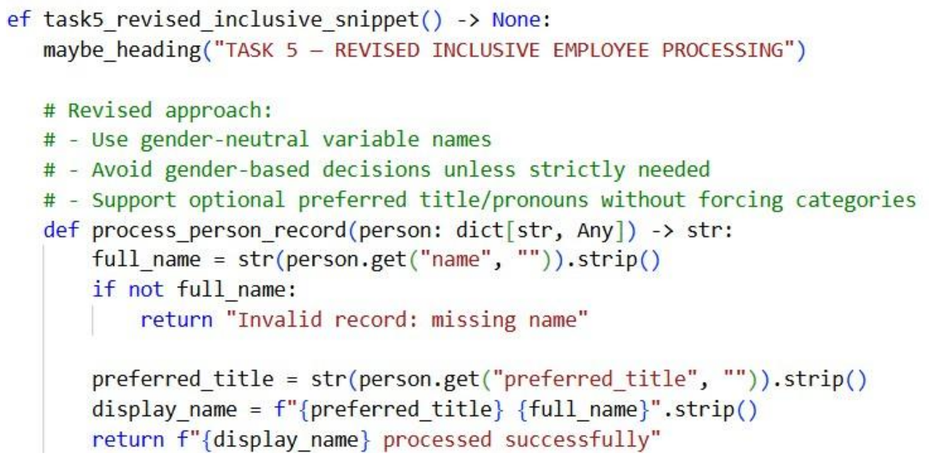
CODE:



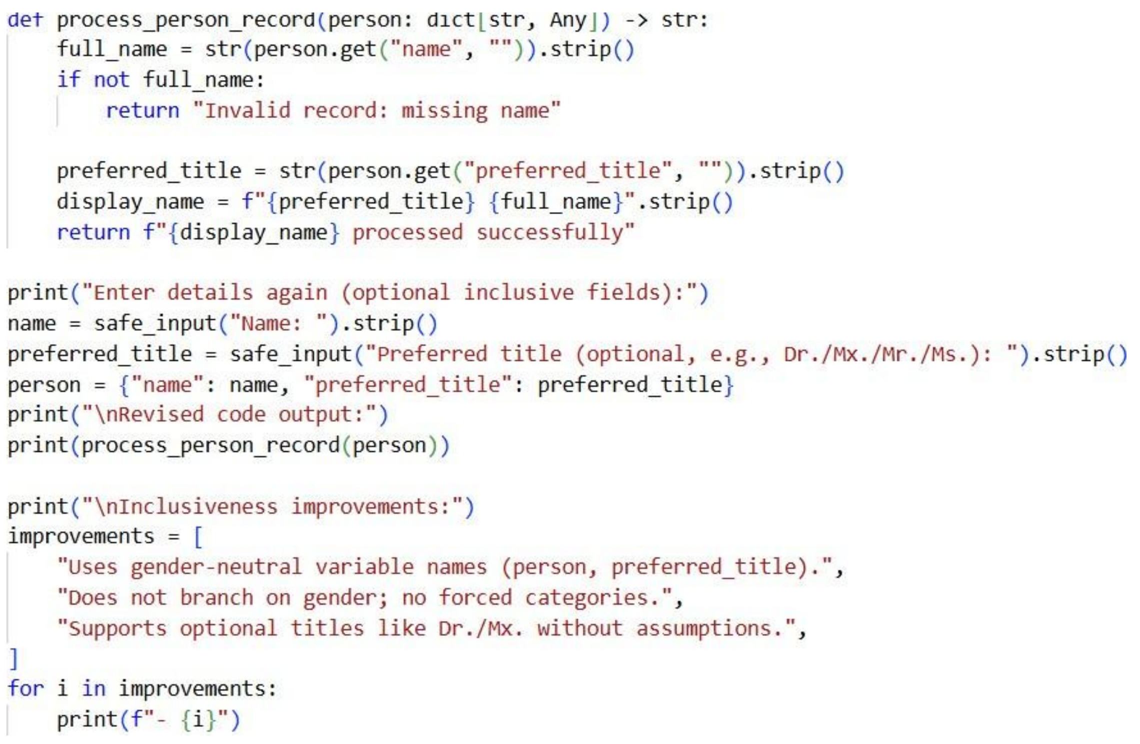
```
# ====================================================================================
# Task 5: Inclusiveness and Ethical Variable Design
# ====================================================================================

def task5_original_non_inclusive_snippet() -> None:
    maybe_heading("TASK 5 – AI-GENERATED EMPLOYEE PROCESSING (NON-INCLUSIVE DEMO)")

    # --- AI-GENERATED (non-inclusive) VERSION ---
    # Intentional issues:
    # - Assumes only male/female
    # - Uses gender-based titles/logic unnecessarily
    def process_employee_original(employee: dict[str, Any]) -> str:
        name = employee["name"]
        gender = employee["gender"]  # expected 'male' or 'female'

        if gender == "male":
            title = "Mr."
        else:
            title = "Ms."  # assumes anyone not 'male' is 'female'

        return f"{title} {name} processed successfully"

    print("Enter person details:")
    name = safe_input("Name: ").strip()
    gender = safe_input("Gender (any text): ").strip()

    example = {"name": name, "gender": gender}
    print("\nOriginal code output:")
    print(process_employee_original(example))
```



```
    print("Enter person details:")
    name = safe_input("Name: ").strip()
    gender = safe_input("Gender (any text): ").strip()

    example = {"name": name, "gender": gender}
    print("\nOriginal code output:")
    print(process_employee_original(example))

    print("\nNon-inclusive issues identified:")
    issues = [
        "Assumes only two genders (male/female).",
        "Defaults everyone else to 'Ms.' which is incorrect and disrespectful.",
        "Uses gender for a title even when title is not required by the task.",
    ]
    for i in issues:
        print(f"- {i}")
```

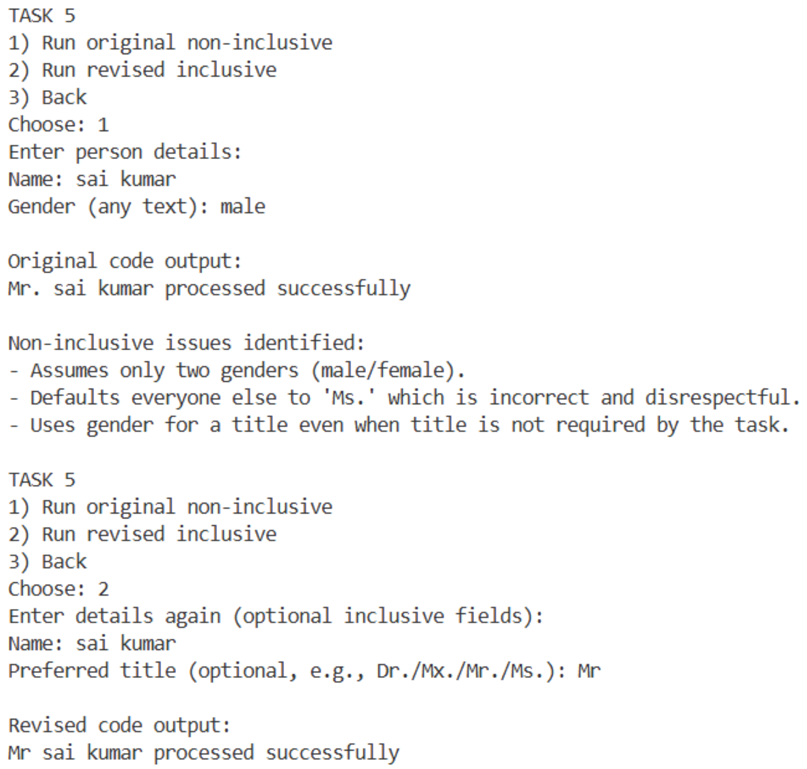


```
ef task5_revised_inclusive_snippet() -> None:
    maybe_heading("TASK 5 — REVISED INCLUSIVE EMPLOYEE PROCESSING")

    # Revised approach:
    # - Use gender-neutral variable names
    # - Avoid gender-based decisions unless strictly needed
    # - Support optional preferred title/pronouns without forcing categories
    def process_person_record(person: dict[str, Any]) -> str:
        full_name = str(person.get("name", "")).strip()
        if not full_name:
            return "Invalid record: missing name"

        preferred_title = str(person.get("preferred_title", "")).strip()
        display_name = f"{preferred_title} {full_name}".strip()
        return f"{display_name} processed successfully"
```



```
    def process_person_record(person: dict[str, Any]) -> str:
        full_name = str(person.get("name", "")).strip()
        if not full_name:
            return "Invalid record: missing name"

        preferred_title = str(person.get("preferred_title", "")).strip()
        display_name = f"{preferred_title} {full_name}".strip()
        return f"{display_name} processed successfully"

    print("Enter details again (optional inclusive fields):")
    name = safe_input("Name: ").strip()
    preferred_title = safe_input("Preferred title (optional, e.g., Dr./Mx./Mr./Ms.): ").strip()
    person = {"name": name, "preferred_title": preferred_title}
    print("\nRevised code output:")
    print(process_person_record(person))

    print("\nInclusiveness improvements:")
    improvements = [
        "Uses gender-neutral variable names (person, preferred_title).",
        "Does not branch on gender; no forced categories.",
        "Supports optional titles like Dr./Mx. without assumptions.",
    ]
    for i in improvements:
        print(f"- {i}")
```

OUTPUT:



```
TASK 5
1) Run original non-inclusive
2) Run revised inclusive
3) Back
Choose: 1
Enter person details:
Name: sai kumar
Gender (any text): male

Original code output:
Mr. sai kumar processed successfully

Non-inclusive issues identified:
- Assumes only two genders (male/female).
- Defaults everyone else to 'Ms.' which is incorrect and disrespectful.
- Uses gender for a title even when title is not required by the task.

TASK 5
1) Run original non-inclusive
2) Run revised inclusive
3) Back
Choose: 2
Enter details again (optional inclusive fields):
Name: sai kumar
Preferred title (optional, e.g., Dr./Mx./Mr./Ms.): Mr

Revised code output:
Mr sai kumar processed successfully
```

JUSTIFICATION:

This task highlights how AI-generated code may unintentionally include non-inclusive assumptions or gender-specific logic. Revising the code promotes inclusive design practices, ethical variable naming, and respectful handling of user information, which are essential for responsible software development.