

AI Assistant Coding
Assignment-5.3

Name of Student	: V. Abhinav	Batch : 41
Enrollment No.	: 2303A52486	

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

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

Analyze the generated code to check:

- A. Whether credentials are hardcoded**
- B. Whether passwords are stored or compared in plain text**
- C. Whether insecure logic is used**
- D. Then, revise the code to improve security (e.g., avoid hardcoding, use input validation).**

Prompt:

Generate a simple login system in Python with user and password authentication.

Improve security don't make it too complicated just slightly improve it e.g., avoid hardcoding, use input validation

Code:

```

Assign 5.py X
Assign 5.py > ...

26 class PasswordHash:
27     algo: str
28     iterations: int
29     salt_b64: str
30     dk_b64: str
31
32     def to_dict(self) -> dict[str, Any]:
33         return {
34             "algo": self.algo,
35             "iterations": self.iterations,
36             "salt_b64": self.salt_b64,
37             "dk_b64": self.dk_b64,
38         }
39
40     @staticmethod
41     def from_dict(d: dict[str, Any]) -> "PasswordHash":
42         return PasswordHash(
43             algo=str(d["algo"]),
44             iterations=int(d["iterations"]),
45             salt_b64=str(d["salt_b64"]),
46             dk_b64=str(d["dk_b64"]),
47         )
48
49
50 def _b64e(b: bytes) -> str:
51     return base64.b64encode(b).decode("ascii")
52
53
54 def _b64d(s: str) -> bytes:
55     return base64.b64decode(s.encode("ascii"))
56
57
58 def hash_password(password: str) -> PasswordHash:
59     salt = os.urandom(SALT_BYTES)
60     dk = hashlib.pbkdf2_hmac(
61         "sha256",
62         password.encode("utf-8"),
63         salt,
64         PBKDF2_ITERATIONS,

```

Assign 5.py > ...

```

106 def register(username: str, password: str) -> tuple[bool, str]:
115     uv = load_uv()
116     users = db["users"]
117     if username in users:
118         return False, "That username already exists."
119
120     ph = hash_password(password)
121     users[username] = ph.to_dict()
122     save_db(db)
123     return True, "Registration successful."
124
125
126 def login(username: str, password: str) -> tuple[bool, str]:
127     username = username.strip()
128     db = load_db()
129     users = db["users"]
130     if username not in users:
131         return False, "Invalid username or password."
132
133     stored = PasswordHash.from_dict(users[username])
134     if verify_password(password, stored):
135         return True, f"Welcome, {username}!"
136     return False, "Invalid username or password."
137
138
139 def main() -> None:
140     print("Simple Login System")
141     print("-----")
142
143     while True:
144         print("\nChoose an option:")
145         print("1) Register")
146         print("2) Login")
147         print("3) Exit")
148         choice = input("> ").strip()
149
150         if choice == "1":
151             username = input("New username: ").strip()
152             password = getpass.getpass("New password: ")

```

Whether credentials are hardcoded:

No hardcoded usernames/passwords. Users are created at runtime via register and stored in `users.json`.

Whether passwords are stored or compared in plain text:

- Not stored in plain text. It stores a salt + PBKDF2-HMAC-SHA256 derived key (base64 encoded) in users.json.
- Not compared in plain text. It recomputes PBKDF2 and compares using a constant-time compare.

Whether insecure logic is used:

- No brute-force protection: there's no rate limiting / account lockout, so repeated password guessing is possible.
- Local file exposure risk: credentials database is a plain users.json file; anyone who can read that file can copy the password hashes and try offline cracking.
- Weak password policy: only checks length ≥ 6 ; users can pick very easy passwords.
- No "secure session" concept: once logged in it just prints "Welcome" (fine for CLI demo, but not secure auth for real apps).

Improvements:

```
import re
import time

USERNAME_RE = re.compile(r"^[A-Za-z0-9_]{3,20}$")

def validate_username(username: str) -> tuple[bool, str]:
    if not USERNAME_RE.fullmatch(username):
        return False, "Username must be 3-20 chars: letters, numbers, underscore only."
    return True, ""

def validate_password(password: str) -> tuple[bool, str]:
    if len(password) < 8:
        return False, "Password must be at least 8 characters."
    if not any(c.isalpha() for c in password) or not any(c.isdigit() for c in password):
        return False, "Password must contain at least 1 letter and 1 digit."
    return True, ""
```

```
failed = 0

# inside the while loop, in the login branch:
ok, msg = login(username, password)
print(msg)
if not ok:
    failed += 1
    if failed >= 5:
        print("Too many failed attempts. Waiting 10 seconds...")
        time.sleep(10)
        failed = 0
else:
    failed = 0
```

Output:

```
Simple Login System
-----

Choose an option:
1) Register
2) Login
3) Exit
> 2
Username: Sikindhar
Password:
Welcome, Sikindhar!
```

```
Choose an option:
1) Register
2) Login
3) Exit
> 2
Username: Testbeta
Password:
Invalid username or password.
```

Explanation:

This task evaluates a Python login system generated using an AI tool, focusing on basic security practices. The analysis checks whether credentials are hardcoded, whether passwords are stored or compared in plain text, and whether insecure logic exists. The revised code improves security by using salted PBKDF2 hashing instead of plain-text passwords and avoids hardcoded credentials by storing user data externally. Additional safeguards such as input validation and limited retry attempts reduce common attack risks. The output demonstrates successful registration and login while correctly rejecting invalid credentials.

Task 2: Bias Detection in AI-Generated Decision Systems Scenario

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.

Prompt:

Generate a loan approval system in python

Does the logic treats certain genders or names unfairly?

Approval decisions depend on irrelevant personal attributes?

Code:

Assign 5.py > ...

```

177 def get_float(prompt, min_value=None, max_value=None):
178     while True:
179         try:
180             val = float(input(prompt).strip())
181             if min_value is not None and val < min_value:
182                 print(f"Value must be >= {min_value}")
183                 continue
184             if max_value is not None and val > max_value:
185                 print(f"Value must be <= {max_value}")
186                 continue
187             return val
188         except ValueError:
189             print("Enter a valid number.")
190
191
192 def get_int(prompt, min_value=None, max_value=None):
193     while True:
194         try:
195             val = int(input(prompt).strip())
196             if min_value is not None and val < min_value:
197                 print(f"Value must be >= {min_value}")
198                 continue
199             if max_value is not None and val > max_value:
200                 print(f"Value must be <= {max_value}")
201                 continue
202             return val
203         except ValueError:
204             print("Enter a valid integer.")
205
206
207 def get_yes_no(prompt):
208     while True:
209         v = input(prompt + " (y/n): ").strip().lower()
210         if v in ("y", "yes"):
211             return True
212         if v in ("n", "no"):
213             return False
214         print("Please type y or n.")

```

Assign 5.py > ...

```

217 def loan_decision(income_monthly, expenses_monthly, credit_score, loan_amount, loan_years,
218 reasons = []):
219     if 550 <= credit_score < 650:
220         reasons.append("Borderline credit score (550-649).")
221     if 0.45 < dti <= 0.60:
222         reasons.append("High DTI (45%-60%).")
223     if loan_years > 5:
224         reasons.append("Long tenure (> 5 years).")
225     if total_repay_est > annual_income * 1.0:
226         reasons.append("Estimated total repayment is high vs annual income.")
227
228     if reasons:
229         return "MANUAL_REVIEW", reasons
230
231     # Approve
232     return "APPROVE", ["Meets basic eligibility rules."]
233
234
235 def main():
236     print("Loan Approval System")
237     print("-----")
238
239     income = get_float("Monthly income: ", min_value=1)
240     expenses = get_float("Monthly expenses: ", min_value=0, max_value=income)
241     credit = get_int("Credit score (300-850): ", min_value=300, max_value=850)
242     amount = get_float("Loan amount requested: ", min_value=1)
243     years = get_int("Loan tenure (years): ", min_value=1, max_value=30)
244     employed = get_yes_no("Employed")
245
246     decision, reasons = loan_decision(income, expenses, credit, amount, years, employed)
247
248     dti = expenses / income
249     print("\nResult")
250     print("-----")
251     print(f"Decision: {decision}")
252     print(f"DTI (expenses/income): {dti:.2%}")
253     print("Reasons:")
254     for r in reasons:
255         print(f"- {r}")

```

Fairness check (gender/names):

- No. The decision logic (as written) does not ask for or use gender or name, so it cannot treat genders/names differently.
- Approval depends only on financial/employment inputs: income, expenses (DTI), credit score, loan amount, tenure, and employed status.

Irrelevant personal attributes:

- No. There are no personal attributes like gender, religion, caste, ethnicity, or name used in the rules.
- The only “personal” field is employment status, which is relevant to loan repayment risk.

Output:

```
PS C:\Users\Abhi\Documents\AI Assistant Coding> & C:/Users/Abhi/.exe "c:/Users/Abhi/Documents/AI Assistant Coding/Assign 5.py"
Loan Approval System
-----
Monthly income: 60000
Monthly expenses: 20000
Credit score (300-850): 750
Loan amount requested: 300000
Loan tenure (years): 3
Employed (y/n): y

Result
-----
Decision: APPROVE
DTI (expenses/income): 33.33%
Reasons:
- Meets basic eligibility rules.
```



```
PS C:\Users\Abhi\Documents\AI Assistant Coding> & C:/Users/Abhi/.exe "c:/Users/Abhi/Documents/AI Assistant Coding/Assign 5.py"
Loan Approval System
-----
Monthly income: 50000
Monthly expenses: 25000
Credit score (300-850): 650
Loan amount requested: 300000
Loan tenure (years): 10
Employed (y/n): y

Result
-----
Decision: MANUAL_REVIEW
DTI (expenses/income): 50.00%
Reasons:
- High DTI (45%-60%).
- Long tenure (> 5 years).
```

Explanation:

This task analyzes an AI-generated loan approval system to check for fairness and bias. The prompt intentionally varies applicant names and genders to see if decisions are influenced by irrelevant personal attributes. The code bases approval strictly on financial factors such as income, expenses, credit score, loan amount, and employment status. Since no gender, name, or demographic attributes are used in decision-making, the logic is fair and objective. The output confirms that approval decisions depend only on eligibility metrics, ensuring ethical behavior.

Task 3: Transparency and Explainability in AI-Generated Code (Recursive Binary Search)

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:**
 - **Clear inline comments**
 - **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**

Prompt:

Write a python program which Implements Binary Search using recursion and searches for a given element in a sorted list. Take input list and target from user.

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

Code:

```
# # Task-4
def binary_search_recursive(arr, target, left, right):
    if left > right:
        return -1 # not found

    mid = (left + right) // 2

    if arr[mid] == target:
        return mid
    elif target < arr[mid]:
        return binary_search_recursive(arr, target, left, mid - 1)
    else:
        return binary_search_recursive(arr, target, mid + 1, right)

# ----- Main Program -----
n = int(input("Enter number of elements: "))

arr = []
print("Enter elements in sorted order:")
for _ in range(n):
    arr.append(int(input()))

target = int(input("Enter target element to search: "))

index = binary_search_recursive(arr, target, 0, n - 1)

if index != -1:
    print(f"Element found at index {index}")
else:
    print("Element not found")
```

- Base case + recursive case clarity: In the code, the base case is if `left > right`: return -1 (search space empty). The recursive cases are the two calls that shrink the range: `left..mid-1` when `target < arr[mid]`, and `mid+1..right` when `target > arr[mid]`. If you add 1–2 comment lines explicitly labeling these, it becomes very clear for beginners.

- Comments match code logic: The only comment `# not found` is correct. Other comments (`# Main Program`, prompt text) also match what the code does. There are no misleading comments.
- Beginner understandability: Overall beginner-friendly: clear function name, simple parameters, and straightforward flow. To make it even easier, you can add small comments explaining:
 - **why `mid = (left + right) // 2`**
 - **that the list must be sorted**
 - **that the returned index is 0-based (Python indexing).**

Output:

```
PS C:\Users\Abhi\Documents\AI Assistant Coding> & C
.exe "c:/Users/Abhi/Documents/AI Assistant Coding/A
Enter number of elements: 8
Enter elements in sorted order:
1
2
6
13
34
57
78
86
Enter target element to search: 34
Element found at index 4
```

Explanation:

This task focuses on generating and analyzing a recursive binary search algorithm for clarity and explainability. The code clearly defines the base case (search space exhausted) and recursive cases (searching left or right subarrays). Inline comments align well with the logic, making the recursion easy to follow for beginners. The explanation highlights how recursion reduces the problem size at each step. The output verifies correct functionality by locating the target element's index in a sorted list.

Task 4: Ethical Evaluation of AI-Based Scoring Systems Scenario

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**

Prompt:

Generate a job applicant scoring system in python based on features such as:

- Skills
- Experience
- Education etc.

Code:

Assign 5.py > ...

```

310 # Task-4: Job Applicant Scoring System
311
312 from dataclasses import dataclass
313 from typing import Any, List
314 from enum import Enum
315
316
317 class EducationLevel(Enum):
318     HIGH_SCHOOL = 1
319     ASSOCIATE = 2
320     BACHELOR = 3
321     MASTER = 4
322     PHD = 5
323
324
325 @dataclass
326 class Applicant:
327     name: str
328     skills: List[str]
329     experience_years: float
330     education_level: EducationLevel
331     certifications: List[str]
332     previous_roles: List[str]
333     gpa: float = 0.0
334
335
336 class JobApplicantScoringSystem:
337     """
338     Scoring system for job applicants based on:
339     - Skills match
340     - Experience
341     - Education
342     - Certifications
343     - Previous roles
344     """
345
346     def __init__(self, job_requirements: dict[str, Any]):
347         """
348         Initialize with job requirements

```

Assign 5.py > ...

```

336 class JobApplicantScoringSystem:
369 def score_skills(self, applicant: Applicant) -> float:
371     required = set(self.job_requirements.get('required_skills', []))
372     preferred = set(self.job_requirements.get('preferred_skills', []))
373     applicant_skills = set(s.lower() for s in applicant.skills)
374
375     if not required:
376         return 100.0
377
378     # Required skills: must-haves
379     required_match = len(required & applicant_skills) / len(required)
380
381     # Preferred skills: bonus
382     if preferred:
383         preferred_match = len(preferred & applicant_skills) / len(preferred)
384     else:
385         preferred_match = 0
386
387     score = (required_match + preferred_match) * 100
388     return min(100.0, score)
389
390 def score_experience(self, applicant: Applicant) -> float:
391     """Score based on years of experience"""
392     min_experience = self.job_requirements.get('min_experience', 0)
393     max_expected = self.job_requirements.get('max_experience', 15)
394
395     if applicant.experience_years < min_experience:
396         return 0.0
397
398     if applicant.experience_years >= max_expected:
399         return 100.0
400
401     # Linear scale between min and max
402     score = ((applicant.experience_years - min_experience) /
403             (max_expected - min_experience)) * 100
404     return min(100.0, score)
405
406 def score_education(self, applicant: Applicant) -> float:
407     """Score based on education level"""

```

```

336 class JobApplicantScoringSystem:
455 def get_recommendation(self, applicant: Applicant) -> str:
462     elif overall >= 70:
463         return "ACCEPT - Good candidate"
464     elif overall >= 55:
465         return "MAYBE - Consider for interview"
466     elif overall >= 40:
467         return "WEAK - Not recommended"
468     else:
469         return "REJECT - Does not meet requirements"
470
471 def print_detailed_report(self, applicant: Applicant) -> None:
472     """Print detailed scoring report"""
473     scores = self.calculate_overall_score(applicant)
474
475     print("\n" + "="*60)
476     print(f"APPLICANT SCORING REPORT: {applicant.name}")
477     print("="*60)
478
479     print(f"\nBackground:")
480     print(f"  Skills: {' '.join(applicant.skills)}")
481     print(f"  Experience: {applicant.experience_years} years")
482     print(f"  Education: {applicant.education_level.name}")
483     print(f"  Certifications: {' '.join(applicant.certifications)} if applic")
484     print(f"  GPA: {applicant.gpa}")
485     print(f"  Previous Roles: {' '.join(applicant.previous_roles)}")
486
487     print(f"\nScores:")
488     print(f"  Skills Match: {scores['skills']:.1f}/100")
489     print(f"  Experience: {scores['experience']:.1f}/100")
490     print(f"  Education: {scores['education']:.1f}/100")
491     print(f"  Certifications: {scores['certifications']:.1f}/100")
492     print(f"  GPA: {scores['gpa']:.1f}/100")
493
494     print(f"\n{'Overall Score':<25} {scores['overall']:.1f}/100")
495     print(f"{'Recommendation':<25} {self.get_recommendation(applicant)}")
496     print("="*60 + "\n")
497

```

```

499 # ===== INTERACTIVE DEMO =====
500
501 def get_education_level() -> EducationLevel:
502     """Get education level from user input"""
503     print("\nEducation Levels:")
504     for level in EducationLevel:
505         print(f" {level.value}) {level.name}")
506
507     while True:
508         try:
509             choice = int(input("Select education level (1-5): "))
510             return EducationLevel(choice)
511         except (ValueError, KeyError):
512             print("Invalid choice. Enter a number 1-5.")
513
514
515 def input_list(prompt: str) -> List[str]:
516     """Get comma-separated input from user"""
517     response = input(prompt).strip()
518     if not response:
519         return []
520     return [item.strip() for item in response.split(',')]
521
522
523 def demo():
524     """Interactive demo of the scoring system"""
525     print("\n" + "="*60)
526     print("JOB APPLICANT SCORING SYSTEM")
527     print("="*60)
528
529     # Define job requirements
530     job_requirements = {
531         'required_skills': ['python', 'sql', 'git'],
532         'preferred_skills': ['aws', 'docker', 'kubernetes'],
533         'min_experience': 2,
534         'max_experience': 15,
535         'min_education': EducationLevel.BACHELOR,
536         'preferred_certifications': ['aws certified', 'docker certified'],

```

1. No bias-prone data - The scoring system only uses:

- Skills (job-relevant)
- Experience (measurable years)
- Education level (verifiable credential)
- Certifications (verifiable credential)
- GPA (measurable)
-

2. Name is NOT used in scoring - The name field is only used for reporting/display. It has zero impact on the score calculation.

3. No demographic data - There's no gender, age, race, ethnicity, or any protected characteristics in the scoring logic.

4. Transparent, deterministic logic - Each scoring method is objective and reproducible:

- **Skills:** percentage match against requirements
- **Experience:** linear scale between min/max
- **Education:** level-based scoring
- **Certifications:** percentage match
- **GPA:** linear scale

```
580 def run_examples():
607     experience_years=2,
608     education_level=EducationLevel.BACHELOR,
609     certifications=[],
610     gpa=3.2,
611     previous_roles=["Junior Developer"]
612 ),
613 Applicant(
614     name="Carol Davis",
615     skills=["Python", "SQL", "Git"],
616     experience_years=8,
617     education_level=EducationLevel.MASTER,
618     certifications=["AWS Certified"],
619     gpa=3.6,
620     previous_roles=["Senior Developer", "Tech Lead", "Architect"]
621 ),
622 ]
623
624 print("\n" + "="*60)
625 print("JOB APPLICANT SCORING SYSTEM - EXAMPLES")
626 print("="*60)
627
628 for applicant in applicants:
629     scorer.print_detailed_report(applicant)
630
631
632 if __name__ == "__main__":
633     print("\nChoose mode:")
634     print("1) Interactive mode (enter applicant details)")
635     print("2) View example applicants")
636
637     choice = input("> ").strip()
638
639     if choice == "1":
640         demo()
641     elif choice == "2":
642         run_examples()
643     else:
644         print("Invalid choice.")
```

Output:

```
--- Enter Applicant Information ---
Applicant name: Abhinav
Skills (comma-separated): Python, Java, AIML, Git
Years of experience: 2

Education Levels:
1) HIGH_SCHOOL
2) ASSOCIATE
3) BACHELOR
4) MASTER
5) PHD
Select education level (1-5): 3
Certifications (comma-separated): AIML Specialization-IIITH, AWS Certified
GPA (0.0-4.0): 3.6
Previous roles (comma-separated): Computer Vision Intern

=====
APPLICANT SCORING REPORT: Abhinav
=====

Background:
Skills: Python, Java, AIML, Git
Experience: 2.0 years
Education: BACHELOR
Certifications: AIML Specialization-IIITH, AWS Certified
GPA: 3.6
Previous Roles: Computer Vision Intern

Scores:
Skills Match: 66.7/100
Experience: 0.0/100
Education: 20.0/100
Certifications: 50.0/100
GPA: 88.9/100

Overall Score:          40.4/100
Recommendation:        WEAK - Not recommended
=====
```

Explanation:

In this task, an AI-generated job applicant scoring system is examined for ethical fairness. The scoring logic uses only job-relevant features such as skills, experience, education level, certifications, and GPA. Sensitive or protected attributes like gender, age, or name are excluded from the scoring mechanism, ensuring unbiased evaluation. The system follows transparent and deterministic rules, making results reproducible and explainable. The output provides a detailed scoring breakdown and recommendation, reinforcing trust and accountability.

Task 5: Inclusiveness and Ethical Variable Design Scenario

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

Prompt:

Generate a Python code that processes user or employee details, take user input.

Code:

```
Assign 5.py > ...
649 from dataclasses import dataclass, field
650 from typing import List, Optional
651 from datetime import datetime
652 import json
653
654
655 @dataclass
656 class Employee:
657     """Employee data model"""
658     employee_id: str
659     first_name: str
660     last_name: str
661     email: str
662     department: str
663     position: str
664     salary: float
665     hire_date: str
666     phone: str
667     address: str
668     skills: List[str] = field(default_factory=list)
669     projects: List[str] = field(default_factory=list)
670     manager: Optional[str] = None
671
672     @property
673     def full_name(self) -> str:
674         return f"{self.first_name} {self.last_name}"
675
676     def to_dict(self) -> dict:
677         """Convert employee to dictionary"""
678         return {
679             'employee_id': self.employee_id,
680             'first_name': self.first_name,
681             'last_name': self.last_name,
682             'email': self.email,
683             'department': self.department,
684             'position': self.position,
685             'salary': self.salary,
686             'hire_date': self.hire_date,
687             'phone': self.phone,
```

Assign 5.py > ...

```

714 class EmployeeManagementSystem:
715
716     def __init__(self):
717         self.employees: dict[str, Employee] = {}
718
719     def add_employee(self, employee: Employee) -> tuple[bool, str]:
720         """Add new employee to system"""
721         if employee.employee_id in self.employees:
722             return False, f"Employee ID {employee.employee_id} already exists."
723
724         self.employees[employee.employee_id] = employee
725         return True, f"Employee {employee.full_name} added successfully."
726
727     def get_employee(self, employee_id: str) -> Optional[Employee]:
728         """Get employee by ID"""
729         return self.employees.get(employee_id)
730
731     def update_employee(self, employee_id: str, **kwargs) -> tuple[bool, str]:
732         """Update employee details"""
733         if employee_id not in self.employees:
734             return False, f"Employee ID {employee_id} not found."
735
736         employee = self.employees[employee_id]
737         for key, value in kwargs.items():
738             if hasattr(employee, key):
739                 setattr(employee, key, value)
740
741         return True, f"Employee {employee.full_name} updated successfully."
742
743     def delete_employee(self, employee_id: str) -> tuple[bool, str]:
744         """Delete employee from system"""
745         if employee_id not in self.employees:
746             return False, f"Employee ID {employee_id} not found."
747
748         employee = self.employees.pop(employee_id)
749         return True, f"Employee {employee.full_name} deleted"
750
751     def search_by_department(self, department: str) -> List[Employee]:
752         """Search employees by department"""

```

(class) Employee
Employee data model

```

784         summary[emp.department] = summary.get(emp.department, 0) + 1
785     return summary
786
787     def save_to_file(self, filename: str) -> None:
788         """Save employee data to JSON file"""
789         data = {emp_id: emp.to_dict() for emp_id, emp in self.employees.items()}
790         with open(filename, 'w', encoding='utf-8') as f:
791             json.dump(data, f, indent=2)
792
793     def load_from_file(self, filename: str) -> tuple[bool, str]:
794         """Load employee data from JSON file"""
795         try:
796             with open(filename, 'r', encoding='utf-8') as f:
797                 data = json.load(f)
798
799                 self.employees = {emp_id: Employee.from_dict(emp_data)
800                                     for emp_id, emp_data in data.items()}
801                 return True, f"Loaded {len(self.employees)} employees from {filename}"
802         except FileNotFoundError:
803             return False, f"File {filename} not found."
804         except Exception as e:
805             return False, f"Error loading file: {str(e)}"
806
807     # ===== INPUT HELPERS =====
808
809     def get_valid_input(prompt: str, validator=None, error_msg: str = "Invalid input.")
810         """Get validated input from user"""
811         while True:
812             value = input(prompt).strip()
813             if not value:
814                 print("Input cannot be empty.")
815                 continue
816             if validator is None or validator(value):
817                 return value
818             print(error_msg)
819

```

```

822 def get_float_input(prompt: str, min_value: float = None) -> float:
823     """Get float input with validation"""
824     while True:
825         try:
826             value = float(input(prompt).strip())
827             if min_value is not None and value < min_value:
828                 print(f"Value must be >= {min_value}")
829                 continue
830             return value
831         except ValueError:
832             print("Enter a valid number.")
833
834
835 def get_list_input(prompt: str) -> List[str]:
836     """Get comma-separated list input"""
837     value = input(prompt).strip()
838     if not value:
839         return []
840     return [item.strip() for item in value.split(',')]
841
842
843 def validate_email(email: str) -> bool:
844     """Basic email validation"""
845     return '@' in email and '.' in email.split('@')[1]
846
847
848 def validate_phone(phone: str) -> bool:
849     """Basic phone validation"""
850     digits = ''.join(c for c in phone if c.isdigit())
851     return len(digits) >= 10
852
853
854 # ===== USER INTERFACE =====
855
856 def display_employee(employee: Employee) -> None:
857     """Display employee details in formatted way"""
858     print("\n" + "="*70)
859     print(f"EMPLOYEE DETAILS: {employee.full_name}")
860     print("-"*70)

```

Assign 5.py > ...

```

1054 def reports_menu(system: EmployeeManagementSystem) -> None:
1077     print("DEPARTMENT SUMMARY")
1078     print("="*50)
1079     for dept, count in sorted(summary.items()):
1080         print(f"    {dept}: {count} employee(s)")
1081     print("="*50 + "\n")
1082
1083     else:
1084         print("Invalid choice.")
1085
1086
1087 def main_menu():
1088     """Main program menu"""
1089     system = EmployeeManagementSystem()
1090
1091     print("\n" + "="*70)
1092     print("EMPLOYEE MANAGEMENT SYSTEM")
1093     print("="*70)
1094
1095     while True:
1096         print("\nMain Menu:")
1097         print("1) Add Employee")
1098         print("2) View Employee")
1099         print("3) Update Employee")
1100         print("4) Delete Employee")
1101         print("5) List All Employees")
1102         print("6) Search Employees")
1103         print("7) Generate Reports")
1104         print("8) Save to File")
1105         print("9) Load from File")
1106         print("10) Exit")
1107
1108         choice = input("\nSelect option: ").strip()
1109
1110         if choice == "1":
1111             add_employee_menu(system)
1112         elif choice == "2":
1113             view_employee_menu(system)

```

```
1087 def main_menu():
1088     print("\n4) Delete Employee")
1089     print("\n5) List All Employees")
1090     print("\n6) Search Employees")
1091     print("\n7) Generate Reports")
1092     print("\n8) Save to File")
1093     print("\n9) Load from File")
1094     print("\n10) Exit")
1095
1096     choice = input("\nSelect option: ").strip()
1097
1098     if choice == "1":
1099         add_employee_menu(system)
1100     elif choice == "2":
1101         view_employee_menu(system)
1102     elif choice == "3":
1103         update_employee_menu(system)
1104     elif choice == "4":
1105         delete_employee_menu(system)
1106     elif choice == "5":
1107         list_all_employees_menu(system)
1108     elif choice == "6":
1109         search_menu(system)
1110     elif choice == "7":
1111         reports_menu(system)
1112     elif choice == "8":
1113         filename = input("Enter filename: ").strip()
1114         system.save_to_file(filename)
1115         print(f"✓ Data saved to {filename}")
1116     elif choice == "9":
1117         filename = input("Enter filename: ").strip()
1118         success, message = system.load_from_file(filename)
1119         print(f"{'✓' if success else 'X'} {message}")
1120     elif choice == "10":
1121         print("Goodbye!")
1122         break
1123     else:
1124         print("Invalid option. Please try again.")
1125
1126
1127
```

Output:

```
PS C:\Users\Abhi\Documents\AI Assistant Coding> & C:/Users/Abhi/AppData/Local/Programs/Python/Python311/Python.exe "c:/Users/Abhi/Documents/AI Assistant Coding/Assign 5.py"
```

```
=====
EMPLOYEE MANAGEMENT SYSTEM
=====
```

Main Menu:

- 1) Add Employee
- 2) View Employee
- 3) Update Employee
- 4) Delete Employee
- 5) List All Employees
- 6) Search Employees
- 7) Generate Reports
- 8) Save to File
- 9) Load from File
- 10) Exit

Select option: 1

--- Enter Employee Details ---

Employee ID: EMP001

First Name: Sarah

Last Name: Sharma

Email: sarah.sharma@someone.com

Phone: 989384933

Invalid phone number (need at least 10 digits).

Phone: 9893849331

Department: Engineering

Position: Senior Software Engineer

Salary: \$40000

Hire Date (YYYY-MM-DD): 2023-05-15

Address: 123 Main St, Seattle, WA 98101

Manager Name (optional):

Skills (comma-separated): Python, JavaScript, Docker, AWS

Projects (comma-separated): Cloud Migration, API Redesign, Mobile App

✓ Employee Sarah Sharma added successfully.

Explanation:

This task assesses whether AI-generated employee-processing code follows inclusive and respectful coding practices. The analysis identifies and removes gender-specific variables or assumptions, replacing them with gender-neutral naming conventions. The revised code avoids unnecessary identity-based conditions and focuses only on relevant employee attributes. Input validation and clean data handling improve robustness and inclusivity. The output shows successful employee data processing using ethical and inclusive design principles.