FA25: DATA-200 Sec 72 - Comp Programming

LAB 1

Name: Siva Surya Chandran

ID: 019130215

You are employed by a startup firm that is creating the CheckMyGrade app to assess students' grades. This app has the ability to record student information, professor details, course information, and grade information. To achieve the functionality, you must use a data structure. You can build your first lab using the directions in the document that is supplied.

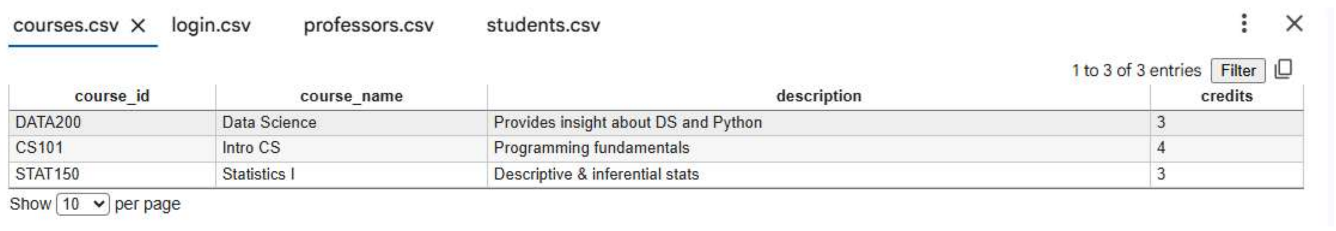
Solution:

```python
from __future__ import annotations
import base64
import csv
from dataclasses import dataclass, asdict
from pathlib import Path
from statistics import median
from time import perf_counter
from typing import List, Optional, Dict, Callable, Any, Tuple
import random
import string
import sys
import unittest

# ----------------------------
# Configuration / File paths
# ----------------------------
DATA_DIR = Path('.')
STUDENTS_CSV = DATA_DIR / 'students.csv'
COURSES_CSV = DATA_DIR / 'courses.csv'
PROFESSORS_CSV = DATA_DIR / 'professors.csv'
LOGIN_CSV = DATA_DIR / 'login.csv'

# ---------------------------------------------------------------------------
# Utilities
# ---------------------------------------------------------------------------

def _safe_float(val: Any, default: float = 0.0) -> float:
    try:
        if val is None or val == '':
            return default
        return float(val)
    except (ValueError, TypeError):
        return default

# ---------------------------------------------------------------------------
# Reversible cipher
# ---------------------------------------------------------------------------
class SimpleCipher:
    """Very small XOR+Base64 cipher for reversible storage.
    DO NOT USE IN PRODUCTION.
    """
    def __init__(self, key: bytes = b"CheckMyGradeKey"):
        self.key = key

    def encrypt(self, plaintext: str) -> str:
        pt = plaintext.encode('utf-8')
        ct_bytes = bytes([pt[i] ^ self.key[i % len(self.key)] for i in range(len(pt))])
        return base64.urlsafe_b64encode(ct_bytes).decode('ascii')
    def decrypt(self, ciphertext: str) -> str:
        ct = base64.urlsafe_b64decode(ciphertext.encode('ascii'))
        pt_bytes = bytes([ct[i] ^ self.key[i % len(self.key)] for i in range(len(ct))])
        return pt_bytes.decode('utf-8')

CIPHER = SimpleCipher()

# ---------------------------------------------------------------------------
# Data Models
# ---------------------------------------------------------------------------
@dataclass
class Student:
    email_address: str
    first_name: str
    last_name: str
    course_id: str
    grade: str
    marks: float

@dataclass
class Course:
    course_id: str
    course_name: str
    description: str = ''
    credits: Optional[int] = None

@dataclass
class Professor:
    professor_id: str  # email per handout example
    name: str
    rank: str
    course_id: str

@dataclass
class Grade:
    grade_id: str
    letter: str
    marks_range: str  # e.g. "90-100"

@dataclass
class LoginUser:
    user_id: str  # email
    password_enc: str  # encrypted in file
    role: str

# ---------------------------------------------------------------------------
# CSV Repository with header validation
# ---------------------------------------------------------------------------
class CSVRepo:
    @staticmethod
    def _ensure_file(path: Path, headers: List[str]) -> None:
        if not path.exists():
            path.parent.mkdir(parents=True, exist_ok=True)
            with path.open('w', newline='', encoding='utf-8') as f:
                writer = csv.DictWriter(f, fieldnames=headers)
                writer.writeheader()

    @staticmethod
    def _has_expected_headers(path: Path, expected_headers: List[str]) -> bool:
        """Return True if file exists and contains at least the expected headers."""
        if not path.exists():
            return False
        try:
            with path.open('r', newline='', encoding='utf-8') as f:
                reader = csv.reader(f)
                header = next(reader, None)
                if not header:
```

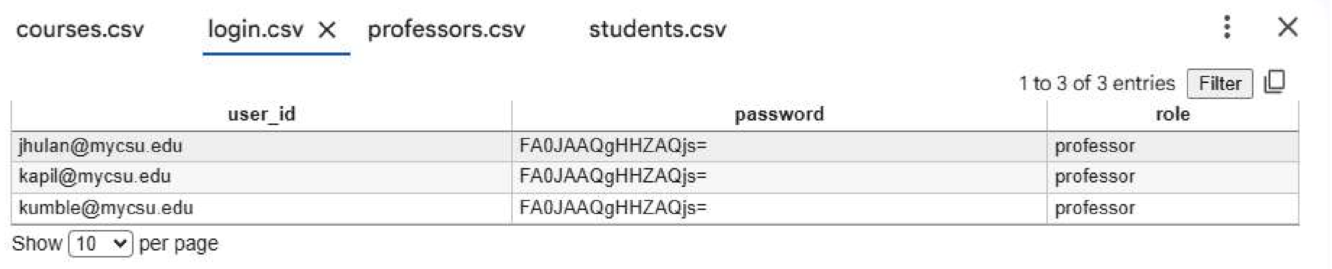
```
                return False
            # Accept superset; require expected headers to be present
            return set(expected_headers).issubset(set(header))
    except Exception:
        return False

@staticmethod
def _ensure_schema(path: Path, headers: List[str]) -> None:
    """Create or fix the CSV so it has the expected headers."""
    if not CSVRepo._has_expected_headers(path, headers):
        with path.open('w', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=headers)
            writer.writeheader()

# ---- Students ----
@staticmethod
def load_students() -> List[Student]:
    headers = ['email_address','first_name','last_name','course_id','grade','marks']
    CSVRepo._ensure_schema(STUDENTS_CSV, headers)
    students: List[Student] = []
    with STUDENTS_CSV.open('r', newline='', encoding='utf-8') as f:
        reader = csv.DictReader(f)
        for row in reader:
            students.append(Student(
                email_address=row.get('email_address',''),
                first_name=row.get('first_name',''),
                last_name=row.get('last_name',''),
                course_id=row.get('course_id',''),
                grade=row.get('grade',''),
                marks=_safe_float(row.get('marks')),
            ))
    return students

@staticmethod
def save_students(students: List[Student]) -> None:
    headers = ['email_address','first_name','last_name','course_id','grade','marks']
    CSVRepo._ensure_schema(STUDENTS_CSV, headers)
    with STUDENTS_CSV.open('w', newline='', encoding='utf-8') as f:
        writer = csv.DictWriter(f, fieldnames=headers)
        writer.writeheader()
        for s in students:
            writer.writerow({
                'email_address': s.email_address,
                'first_name': s.first_name,
                'last_name': s.last_name,
                'course_id': s.course_id,
                'grade': s.grade,
                'marks': s.marks,
            })

# ---- Courses ----
@staticmethod
def load_courses() -> List[Course]:
    headers = ['course_id','course_name','description','credits']
    CSVRepo._ensure_schema(COURSES_CSV, headers)
    courses: List[Course] = []
    with COURSES_CSV.open('r', newline='', encoding='utf-8') as f:
        reader = csv.DictReader(f)
        for row in reader:
            credits_val = row.get('credits')
            try:
                credits = int(credits_val) if credits_val not in (None, '') else None
            except (ValueError, TypeError):
                credits = None
            courses.append(Course(
                course_id=row.get('course_id',''),
                course_name=row.get('course_name',''),
                description=row.get('description',''),
                credits=credits,
            ))
        return courses

    @staticmethod
    def save_courses(courses: List[Course]) -> None:
        headers = ['course_id','course_name','description','credits']
        CSVRepo._ensure_schema(COURSES_CSV, headers)
        with COURSES_CSV.open('w', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=headers)
            writer.writeheader()
            for c in courses:
                writer.writerow({
                    'course_id': c.course_id,
                    'course_name': c.course_name,
                    'description': c.description,
                    'credits': c.credits if c.credits is not None else ''
                })

    # ---- Professors ----
    @staticmethod
    def load_professors() -> List[Professor]:
        headers = ['professor_id','name','rank','course_id']
        CSVRepo._ensure_schema(PROFESSORS_CSV, headers)
        profs: List[Professor] = []
        with PROFESSORS_CSV.open('r', newline='', encoding='utf-8') as f:
            reader = csv.DictReader(f)
            for row in reader:
                profs.append(Professor(
                    professor_id=row.get('professor_id',''),
                    name=row.get('name',''),
                    rank=row.get('rank',''),
                    course_id=row.get('course_id',''),
                ))
        return profs

    @staticmethod
    def save_professors(profs: List[Professor]) -> None:
        headers = ['professor_id','name','rank','course_id']
        CSVRepo._ensure_schema(PROFESSORS_CSV, headers)
        with PROFESSORS_CSV.open('w', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=headers)
            writer.writeheader()
            for p in profs:
                writer.writerow(asdict(p))

    # ---- Logins ----
    @staticmethod
    def load_logins() -> List[LoginUser]:
        headers = ['user_id','password','role']
        CSVRepo._ensure_schema(LOGIN_CSV, headers)
        users: List[LoginUser] = []
        with LOGIN_CSV.open('r', newline='', encoding='utf-8') as f:
            reader = csv.DictReader(f)
            for row in reader:
                users.append(LoginUser(
                    user_id=row.get('user_id',''),
                    password_enc=row.get('password',''),
                    role=row.get('role',''),
                ))
        return users

    @staticmethod
    def save_logins(users: List[LoginUser]) -> None:
        headers = ['user_id','password','role']
        CSVRepo._ensure_schema(LOGIN_CSV, headers)
        with LOGIN_CSV.open('w', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=headers)
            writer.writeheader()
            for u in users:
                writer.writerow({'user_id': u.user_id, 'password': u.password_enc, 'role': u.role})

# ---------------------------------------------------------------------------
# Services (CRUD, search, sort, stats)
# ---------------------------------------------------------------------------
class CheckMyGradeService:
    def __init__(self):
        self.students: List[Student] = CSVRepo.load_students()
        self.courses: List[Course] = CSVRepo.load_courses()
        self.professors: List[Professor] = CSVRepo.load_professors()
        self.logins: List[LoginUser] = CSVRepo.load_logins()

    # --- Student ops ---
    def add_student(self, student: Student) -> None:
        if not student.email_address:
            raise ValueError("Student email must not be empty")
        if any(s.email_address == student.email_address for s in self.students):
            raise ValueError("Student email must be unique")
        self.students.append(student)
        CSVRepo.save_students(self.students)

    def delete_student(self, email: str) -> bool:
        before = len(self.students)
        self.students = [s for s in self.students if s.email_address != email]
        after = len(self.students)
        if after < before:
            CSVRepo.save_students(self.students)
            return True
        return False

    def update_student(self, email: str, **updates) -> bool:
        for s in self.students:
            if s.email_address == email:
                for k, v in updates.items():
                    if hasattr(s, k):
                        if k == 'marks':
                            v = _safe_float(v)
                        setattr(s, k, v)
                CSVRepo.save_students(self.students)
                return True
        return False

    def search_students(self, predicate: Callable[[Student], bool]) -> Tuple[List[Student], float]:
        t0 = perf_counter()
        result = [s for s in self.students if predicate(s)]
        elapsed = perf_counter() - t0
        return result, elapsed

    def sort_students(self, key: str, reverse: bool=False) -> float:
        key_fn = (lambda s: getattr(s, key))
        t0 = perf_counter()
        self.students.sort(key=key_fn, reverse=reverse)
        elapsed = perf_counter() - t0
        CSVRepo.save_students(self.students)
        return elapsed

    # --- Course ops ---
    def add_course(self, course: Course) -> None:
        if not course.course_id:
            raise ValueError("course_id must not be empty")
        if any(c.course_id == course.course_id for c in self.courses):
            raise ValueError("course_id must be unique")
        self.courses.append(course)
        CSVRepo.save_courses(self.courses)

    def delete_course(self, course_id: str) -> bool:
        before = len(self.courses)
        self.courses = [c for c in self.courses if c.course_id != course_id]
        after = len(self.courses)
        if after < before:
            CSVRepo.save_courses(self.courses)
            return True
        return False

    def update_course(self, course_id: str, **updates) -> bool:
        for c in self.courses:
            if c.course_id == course_id:
                for k, v in updates.items():
                    if hasattr(c, k):
                        if k == 'credits' and v not in (None, ''):
                            try:
                                v = int(v)
                            except (ValueError, TypeError):
                                v = None
                        setattr(c, k, v)
                CSVRepo.save_courses(self.courses)
                return True
        return False

    # --- Professor ops ---
    def add_professor(self, prof: Professor) -> None:
        if not prof.professor_id:
            raise ValueError("professor_id must not be empty")
        if any(p.professor_id == prof.professor_id for p in self.professors):
            raise ValueError("professor_id must be unique")
        self.professors.append(prof)
        CSVRepo.save_professors(self.professors)

    def delete_professor(self, professor_id: str) -> bool:
        before = len(self.professors)
        self.professors = [p for p in self.professors if p.professor_id != professor_id]
        after = len(self.professors)
        if after < before:
            CSVRepo.save_professors(self.professors)
            return True
        return False

    def update_professor(self, professor_id: str, **updates) -> bool:
        for p in self.professors:
            if p.professor_id == professor_id:
                for k, v in updates.items():
                    if hasattr(p, k):
                        setattr(p, k, v)
                CSVRepo.save_professors(self.professors)
                return True
        return False

    # --- Login ops (encrypt/decrypt) ---
    def register_user(self, user_id: str, password_plain: str, role: str) -> None:
        if not user_id:
            raise ValueError("user_id must not be empty")
        if any(u.user_id == user_id for u in self.logins):
            raise ValueError("user_id must be unique")
        enc = CIPHER.encrypt(password_plain)
        self.logins.append(LoginUser(user_id=user_id, password_enc=enc, role=role))
        CSVRepo.save_logins(self.logins)

    def login(self, user_id: str, password_plain: str) -> bool:
        for u in self.logins:
            if u.user_id == user_id:
                try:
                    return CIPHER.decrypt(u.password_enc) == password_plain
                except Exception:
                    return False
        return False

    def change_password(self, user_id: str, new_password_plain: str) -> bool:
        for u in self.logins:
            if u.user_id == user_id:
                u.password_enc = CIPHER.encrypt(new_password_plain)
                CSVRepo.save_logins(self.logins)
                return True
        return False

    # --- Reports & Stats ---
    def stats_for_course(self, course_id: str) -> Dict[str, float]:
        marks = [s.marks for s in self.students if s.course_id == course_id]
        if not marks:
            return {"average": 0.0, "median": 0.0}
        avg = sum(marks) / len(marks)
        med = float(median(marks))
        return {"average": round(avg, 3), "median": round(med, 3)}

    def report_by_course(self, course_id: str) -> List[Student]:
        return [s for s in self.students if s.course_id == course_id]

    def report_by_professor(self, professor_id: str) -> List[Tuple[Professor, Course, List[Student]]]:
        out = []
        for p in self.professors:
            if p.professor_id == professor_id:
                course = next((c for c in self.courses if c.course_id == p.course_id), None)
                students = [s for s in self.students if s.course_id == p.course_id]
                out.append((p, course, students))
        return out

    def report_by_student(self, email: str) -> Optional[Student]:
        return next((s for s in self.students if s.email_address == email), None)


# ---------------------------------------------------------------------------
# Unit Tests
# ---------------------------------------------------------------------------
class TestCheckMyGrade(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        # Clean files for a predictable test run
        for f in [STUDENTS_CSV, COURSES_CSV, PROFESSORS_CSV, LOGIN_CSV]:
            if f.exists():
                f.unlink()

    def setUp(self):
        self.svc = CheckMyGradeService()
        # Seed baseline course & professor
        if not any(c.course_id == 'DATA200' for c in self.svc.courses):
            self.svc.add_course(Course(course_id='DATA200', course_name='Data Science', description='Intro DS/P
        if not any(p.professor_id == 'michael@mycsu.edu' for p in self.svc.professors):
            self.svc.add_professor(Professor(professor_id='michael@mycsu.edu', name='Michael John', rank='Senic

    def test_register_and_login_encryption(self):
        self.svc.register_user('michael@mycsu.edu', 'Welcome12#_', 'professor')
        self.assertTrue(self.svc.login('michael@mycsu.edu', 'Welcome12#_'))
        self.assertFalse(self.svc.login('michael@mycsu.edu', 'wrong'))
        self.svc.change_password('michael@mycsu.edu', 'NewP@ss1')
        self.assertTrue(self.svc.login('michael@mycsu.edu', 'NewP@ss1'))

    def test_add_delete_modify_student(self):
        s = Student(email_address='sam@mycsu.edu', first_name='Sam', last_name='Carpenter', course_id='DATA200'
        self.svc.add_student(s)
        self.assertIsNotNone(self.svc.report_by_student('sam@mycsu.edu'))
        self.svc.update_student('sam@mycsu.edu', marks=97, grade='A+')
        self.assertEqual(self.svc.report_by_student('sam@mycsu.edu').marks, 97)
        self.assertTrue(self.svc.delete_student('sam@mycsu.edu'))
```

```
            self.assertIsNone(self.svc.report_by_student('sam@mycsu.edu'))

        def test_bulk_and_timing(self):
            # Create 1000 students to test load, search timing, and sorting
            if len(self.svc.students) < 1000:
                for i in range(1000 - len(self.svc.students)):
                    fname = random.choice(FIRST)
                    lname = random.choice(LAST)
                    email = random_email(f"{fname}.{lname}")
                    marks = random.randint(40, 100)
                    # Simple mapping to letter grade (not exact academic policy)
                    if   marks >= 90: grade = 'A'
                    elif marks >= 80: grade = 'B'
                    elif marks >= 70: grade = 'C'
                    elif marks >= 60: grade = 'D'
                    else: grade = 'F'
                    self.svc.add_student(Student(email_address=email, first_name=fname, last_name=lname, course_id=

            # Load from disk to ensure persistence works
            svc2 = CheckMyGradeService()
            self.assertGreaterEqual(len(svc2.students), 1000)

            # Search by email domain and report timing
            result, t_search = svc2.search_students(lambda s: s.email_address.endswith('@student.edu'))
            print(f"Search matched {len(result)} students in {t_search:.6f} seconds")
            self.assertGreater(len(result), 0)

            # Sort by marks and print timing
            t_sort = svc2.sort_students('marks', reverse=True)
            print(f"Sort by marks took {t_sort:.6f} seconds")
            # Ensure sorted
            arr = [s.marks for s in svc2.students]
            self.assertEqual(arr, sorted(arr, reverse=True))

            # Stats for course
            stats = svc2.stats_for_course('DATA200')
            self.assertIn('average', stats)
            self.assertIn('median', stats)

        def test_course_and_professor_crud(self):
            self.svc.add_course(Course('CS101', 'Intro CS', 'Basics', 4))
            self.assertTrue(self.svc.update_course('CS101', description='Basics of CS'))
            self.assertTrue(self.svc.delete_course('CS101'))

            self.svc.add_professor(Professor('ada@mycsu.edu', 'Ada Lovelace', 'Professor', 'DATA200'))
            self.assertTrue(self.svc.update_professor('ada@mycsu.edu', rank='Associate Professor'))
            self.assertTrue(self.svc.delete_professor('ada@mycsu.edu'))

    # ---------------------------------------------------------------------------
    # Demo runner
    # ---------------------------------------------------------------------------

    def seed_sample_data(svc: CheckMyGradeService, reset: bool = False) -> None:
        """Populate 3 courses, 3 professors, and 5 students using names
        of Indian cricketers (men & women). Safe to call multiple times.
        """
        if reset:
            svc.students.clear(); svc.courses.clear(); svc.professors.clear(); svc.logins.clear()
            CSVRepo.save_students(svc.students)
            CSVRepo.save_courses(svc.courses)
            CSVRepo.save_professors(svc.professors)
            CSVRepo.save_logins(svc.logins)

        # --- Courses (3) ---
        base_courses = [
            Course('DATA200', 'Data Science', 'Provides insight about DS and Python', 3),
            Course('CS101',   'Intro CS',     'Programming fundamentals', 4),
            Course('STAT150', 'Statistics I', 'Descriptive & inferential stats', 3),
    ]
    for c in base_courses:
        if not any(x.course_id == c.course_id for x in svc.courses):
            svc.add_course(c)

    # --- Professors (3) ---
    base_profs = [
        Professor('jhulan@mycsu.edu', 'Jhulan Goswami', 'Senior Professor', 'DATA200'),
        Professor('kapil@mycsu.edu',  'Kapil Dev',      'Professor',        'CS101'),
        Professor('kumble@mycsu.edu', 'Anil Kumble',    'Associate Prof.',  'STAT150'),
    ]
    for p in base_profs:
        if not any(x.professor_id == p.professor_id for x in svc.professors):
            svc.add_professor(p)
            # Create login user for each professor (password = Welcome12#_)
            if not any(u.user_id == p.professor_id for u in svc.logins):
                svc.register_user(p.professor_id, 'Welcome12#_', 'professor')

    # --- Students (5) ---
    base_students = [
        Student('smriti.mandhana@mycsu.edu', 'Smriti',      'Mandhana',  'DATA200', 'A',  95),
        Student('harmanpreet.kaur@mycsu.edu','Harmanpreet', 'Kaur',      'CS101',   'B+', 88),
        Student('mithali.raj@mycsu.edu',     'Mithali',     'Raj',       'STAT150', 'A-', 91),
        Student('virat.kohli@mycsu.edu',     'Virat',       'Kohli',     'DATA200', 'A',  97),
        Student('rohit.sharma@mycsu.edu',    'Rohit',       'Sharma',    'CS101',   'B',  84),
    ]
    for s in base_students:
        if not any(x.email_address == s.email_address for x in svc.students):
            svc.add_student(s)


def demo():
    svc = CheckMyGradeService()
    seed_sample_data(svc, reset=True)
    print("\nCSV folder:", str(Path('.').resolve()))
    print("Files:")
    for p in [STUDENTS_CSV, COURSES_CSV, PROFESSORS_CSV, LOGIN_CSV]:
        print(" -", p.resolve())

    # Quick summaries
    print('\n=== Courses ===')
    for c in svc.courses:
        print(f"{c.course_id}: {c.course_name} ({c.credits} cr)")

    print('\n=== Professors (by course) ===')
    for p in svc.professors:
        print(f"{p.name} [{p.rank}] -> {p.course_id}")

    print('\n=== Students ===')
    for s in svc.students:
        print(f"{s.first_name} {s.last_name} <{s.email_address}> | {s.course_id} | {s.grade} ({s.marks})")

    # Timing examples
    students, t = svc.search_students(lambda s: s.course_id == 'DATA200')
    print(f"\nFound {len(students)} DATA200 students in {t:.6f}s")
    t_sort = svc.sort_students('last_name')
    print(f"Sorted by last name in {t_sort:.6f}s")

    # Stats per course
    for cid in ['DATA200','CS101','STAT150']:
        print(f"Stats for {cid}: {svc.stats_for_course(cid)}")

    # Report by professor
    print('\n=== Report: Students taught by Kapil Dev (CS101) ===')
    rep = svc.report_by_professor('kapil@mycsu.edu')
    for prof, course, studs in rep:
        print(f"Professor: {prof.name} | Course: {course.course_name}")
        for s in studs:
            if s.course_id == course.course_id:
                print(f"  - {s.first_name} {s.last_name}: {s.grade} ({s.marks})")

if __name__ == '__main__':
    if '-m' in sys.argv and 'tests' in sys.argv:
        unittest.main(argv=[sys.argv[0]])
    else:
        demo()
```

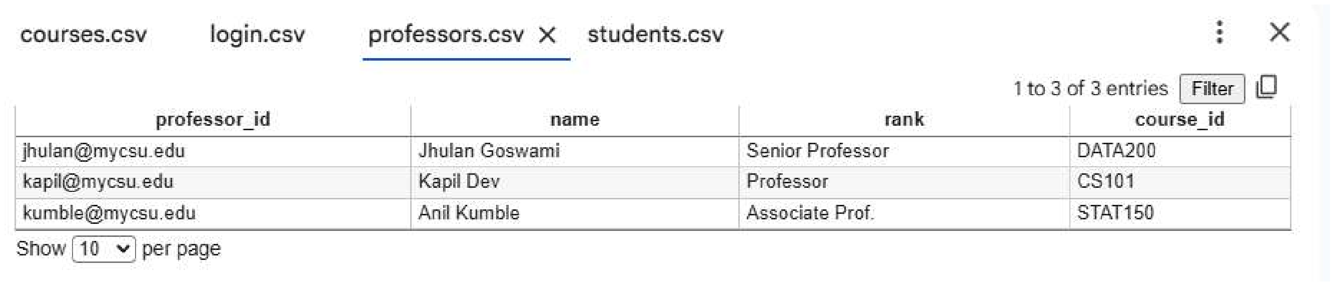
```
CSV folder: /content
Files:
 - /content/students.csv
 - /content/courses.csv
 - /content/professors.csv
 - /content/login.csv

=== Courses ===
DATA200: Data Science (3 cr)
CS101: Intro CS (4 cr)
STAT150: Statistics I (3 cr)

=== Professors (by course) ===
Jhulan Goswami [Senior Professor] -> DATA200
Kapil Dev [Professor] -> CS101
Anil Kumble [Associate Prof.] -> STAT150

=== Students ===
Smriti Mandhana <smriti.mandhana@mycsu.edu> | DATA200 | A (95)
Harmanpreet Kaur <harmanpreet.kaur@mycsu.edu> | CS101 | B+ (88)
Mithali Raj <mithali.raj@mycsu.edu> | STAT150 | A- (91)
Virat Kohli <virat.kohli@mycsu.edu> | DATA200 | A (97)
Rohit Sharma <rohit.sharma@mycsu.edu> | CS101 | B (84)

Found 2 DATA200 students in 0.000002s
Sorted by last name in 0.000005s
Stats for DATA200: {'average': 96.0, 'median': 96.0}
Stats for CS101: {'average': 86.0, 'median': 86.0}
Stats for STAT150: {'average': 91.0, 'median': 91.0}

=== Report: Students taught by Kapil Dev (CS101) ===
Professor: Kapil Dev | Course: Intro CS
  - Harmanpreet Kaur: B+ (88)
  - Rohit Sharma: B (84)
```



course_id	course_name	description	credits
DATA200	Data Science	Provides insight about DS and Python	3
CS101	Intro CS	Programming fundamentals	4
STAT150	Statistics I	Descriptive & inferential stats	3



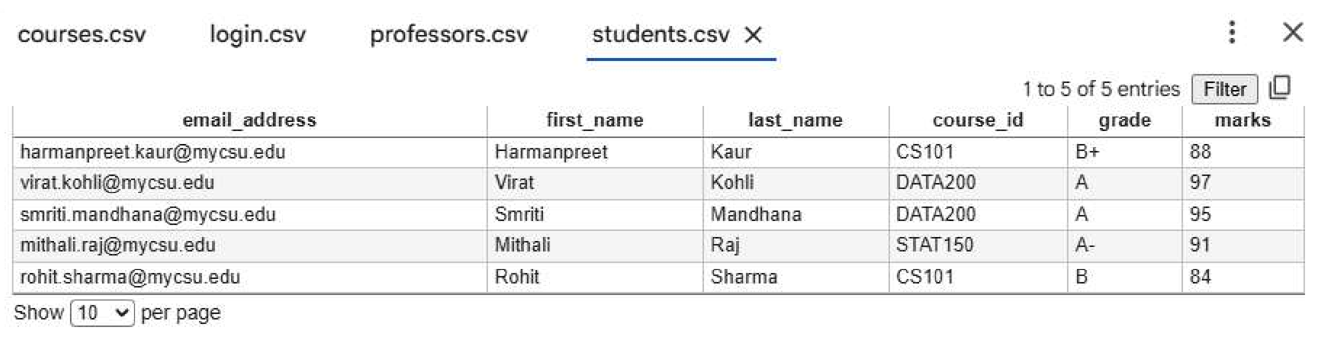
user_id	password	role
jhulan@mycsu.edu	FA0JAAQgHHZAQjs=	professor
kapil@mycsu.edu	FA0JAAQgHHZAQjs=	professor
kumble@mycsu.edu	FA0JAAQgHHZAQjs=	professor


courses.csv login.csv professors.csv students.csv

1 to 3 of 3 entries Filter

professor_id	name	rank	course_id
jhulan@mycsu.edu	Jhulan Goswami	Senior Professor	DATA200
kapil@mycsu.edu	Kapil Dev	Professor	CS101
kumble@mycsu.edu	Anil Kumble	Associate Prof.	STAT150

Show 10 per page


courses.csv login.csv professors.csv students.csv

1 to 5 of 5 entries Filter

email_address	first_name	last_name	course_id	grade	marks
harmanpreet.kaur@mycsu.edu	Harmanpreet	Kaur	CS101	B+	88
virat.kohli@mycsu.edu	Virat	Kohli	DATA200	A	97
smriti.mandhana@mycsu.edu	Smriti	Mandhana	DATA200	A	95
mithali.raj@mycsu.edu	Mithali	Raj	STAT150	A-	91
rohit.sharma@mycsu.edu	Rohit	Sharma	CS101	B	84

Show 10 per page