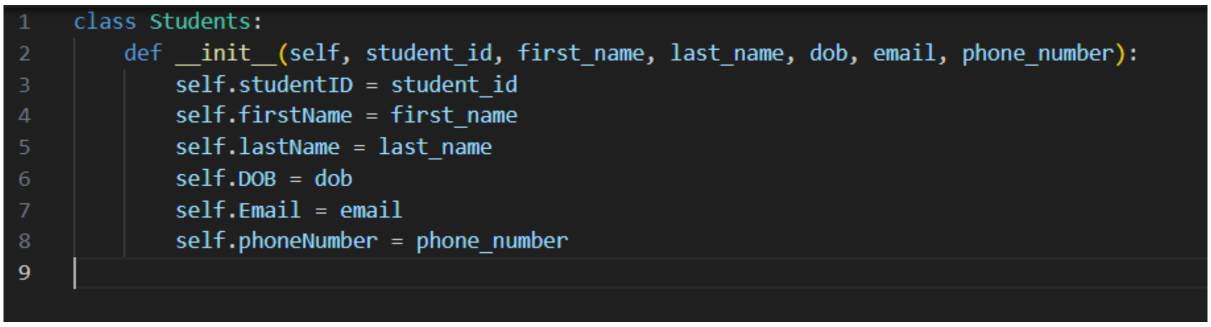
Assignment 2

Student Information System(SIS)

Task 1: Define Classes

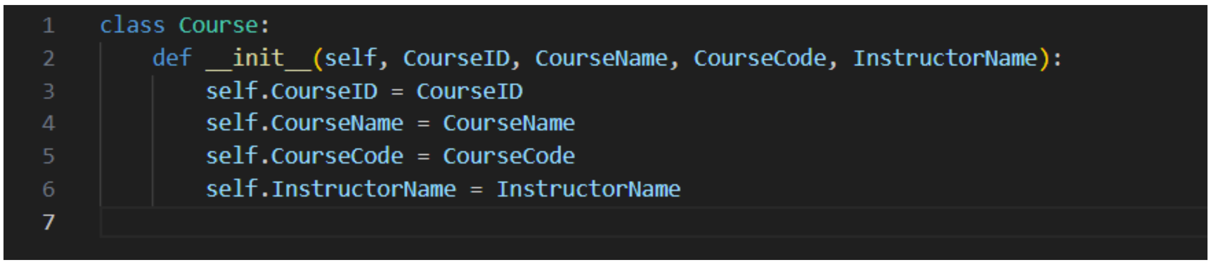
Task 2: Implement Constructors

- Define the following classes based on the domain description:
  Student class with the following attributes:
  - Student ID
  - First Name
  - Last Name
  - Date of Birth
  - Email
  - Phone Number



```python
class Students:
    def __init__(self, student_id, first_name, last_name, dob, email, phone_number):
        self.studentID = student_id
        self.firstName = first_name
        self.lastName = last_name
        self.DOB = dob
        self.Email = email
        self.phoneNumber = phone_number

```

- Course class with the following attributes:
  - Course ID
  - Course Name
  - Course Code
  - Instructor Name



```python
class Course:
    def __init__(self, CourseID, CourseName, CourseCode, InstructorName):
        self.CourseID = CourseID
        self.CourseName = CourseName
        self.CourseCode = CourseCode
        self.InstructorName = InstructorName

```

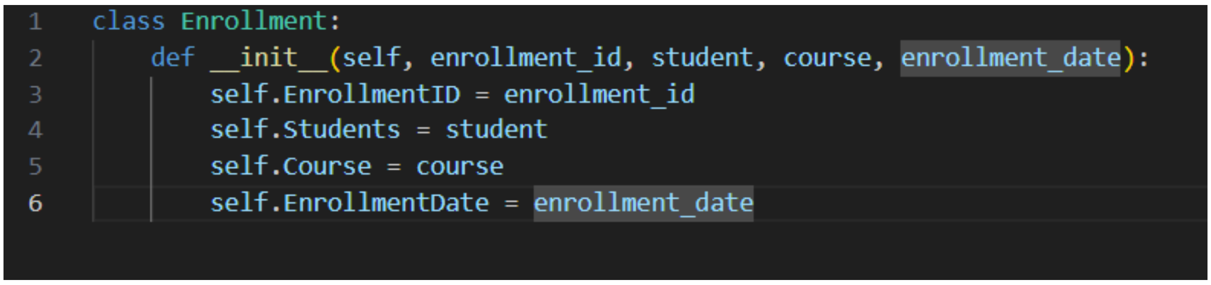
- Enrollment class to represent the relationship between students and courses. It should have attributes:

  • Enrollment ID

  • Student ID (reference to a Student)

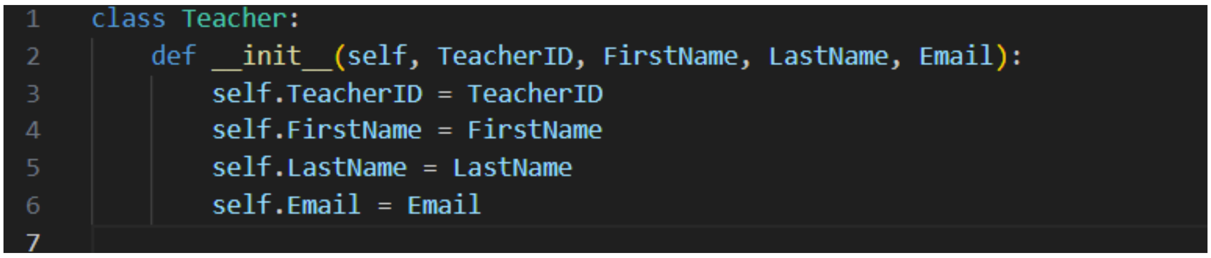
  • Course ID (reference to a Course)

  • Enrollment Date



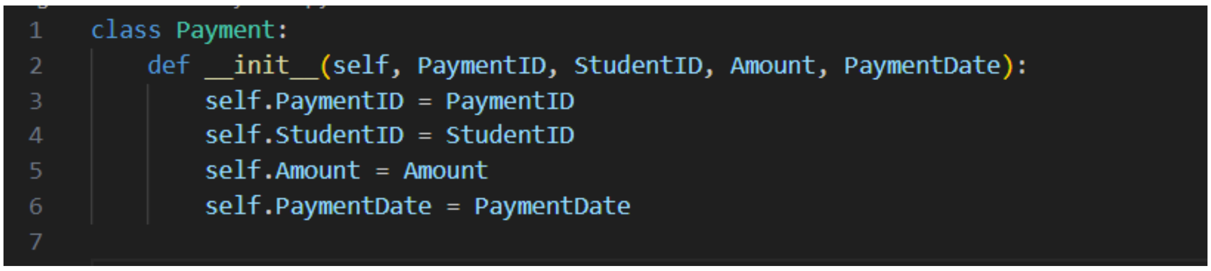
```python
class Enrollment:
    def __init__(self, enrollment_id, student, course, enrollment_date):
        self.EnrollmentID = enrollment_id
        self.Students = student
        self.Course = course
        self.EnrollmentDate = enrollment_date
```

- Teacher class with the following attributes:
    - Teacher ID
    - First Name
    - Last Name
    - Email



```python
class Teacher:
    def __init__(self, TeacherID, FirstName, LastName, Email):
        self.TeacherID = TeacherID
        self.FirstName = FirstName
        self.LastName = LastName
        self.Email = Email

```

- Payment class with the following attributes:
    - Payment ID
    - Student ID (reference to a Student)
    - Amount
    - Payment Date

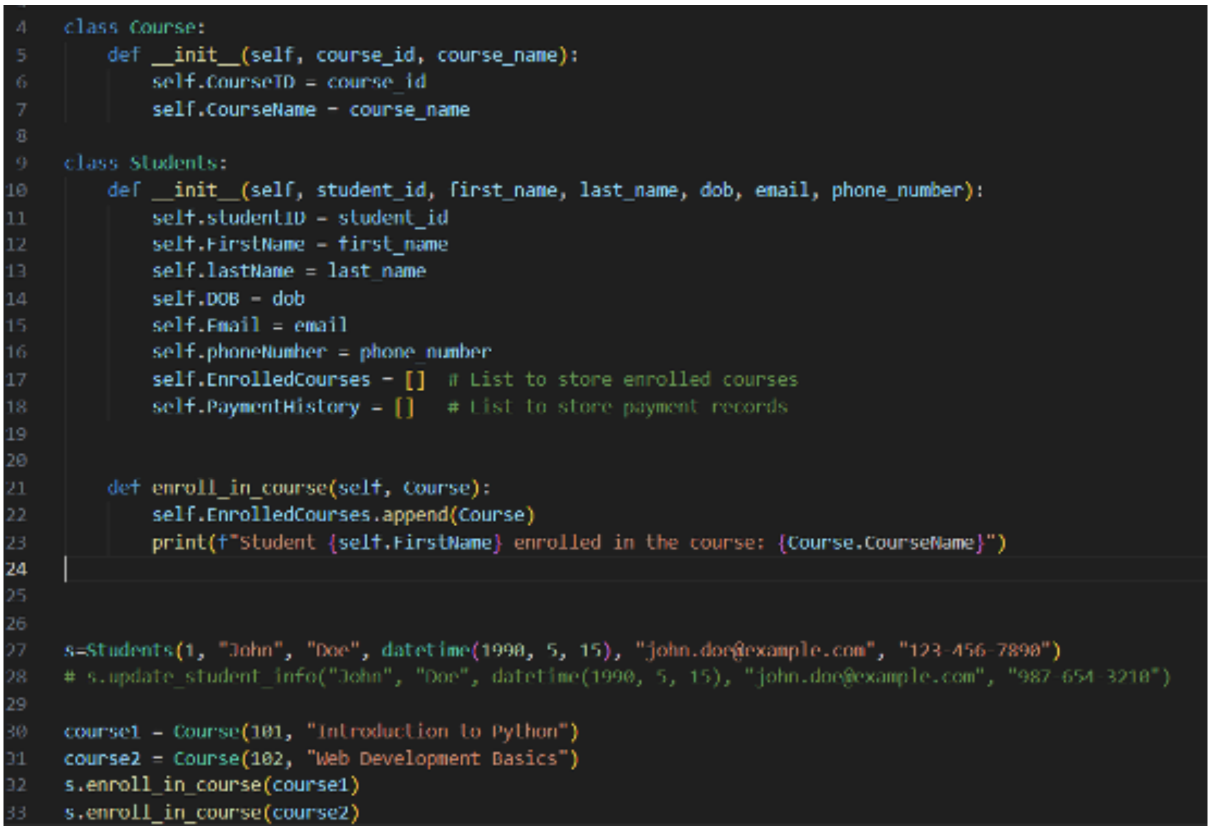


```python
class Payment:
    def __init__(self, PaymentID, StudentID, Amount, PaymentDate):
        self.PaymentID = PaymentID
        self.StudentID = StudentID
        self.Amount = Amount
        self.PaymentDate = PaymentDate

```

Task 3: Implement Methods

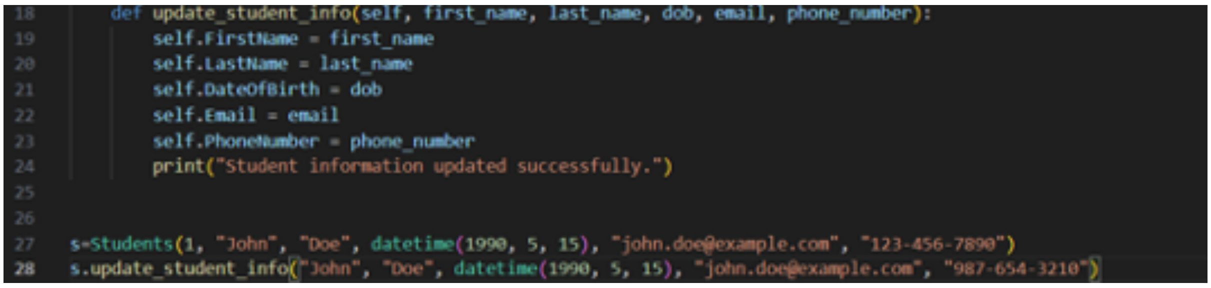
- **Student Class:**

• EnrollInCourse(course: Course): Enrolls the student in a course.



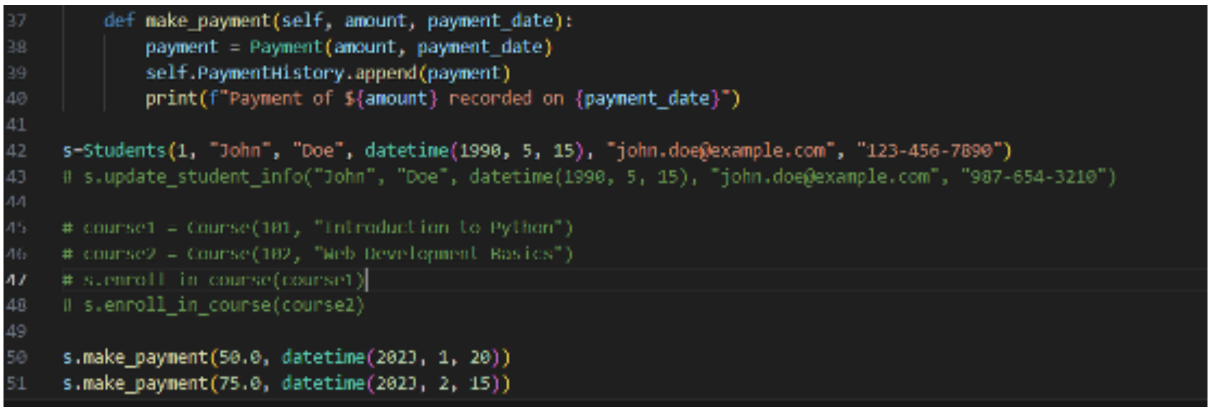
```python
class Course:
    def __init__(self, course_id, course_name):
        self.CourseID = course_id
        self.CourseName = course_name

class Students:
    def __init__(self, student_id, first_name, last_name, dob, email, phone_number):
        self.studentID = student_id
        self.FirstName = first_name
        self.lastName = last_name
        self.DOB = dob
        self.Email = email
        self.phoneNumber = phone_number
        self.EnrolledCourses = []  # List to store enrolled courses
        self.PaymentHistory = []   # List to store payment records


    def enroll_in_course(self, Course):
        self.EnrolledCourses.append(Course)
        print(f"Student {self.FirstName} enrolled in the course: {Course.CourseName}")



s=Students(1, "John", "Doe", datetime(1990, 5, 15), "john.doe@example.com", "123-456-7890")
# s.update_student_info("John", "Doe", datetime(1990, 5, 15), "john.doe@example.com", "987-654-3210")

course1 = Course(101, "Introduction to Python")
course2 = Course(102, "Web Development Basics")
s.enroll_in_course(course1)
s.enroll_in_course(course2)
```

• UpdateStudentInfo(firstName: string, lastName: string, dateOfBirth: DateTime, email: string, phoneNumber: string): Updates the student's information.



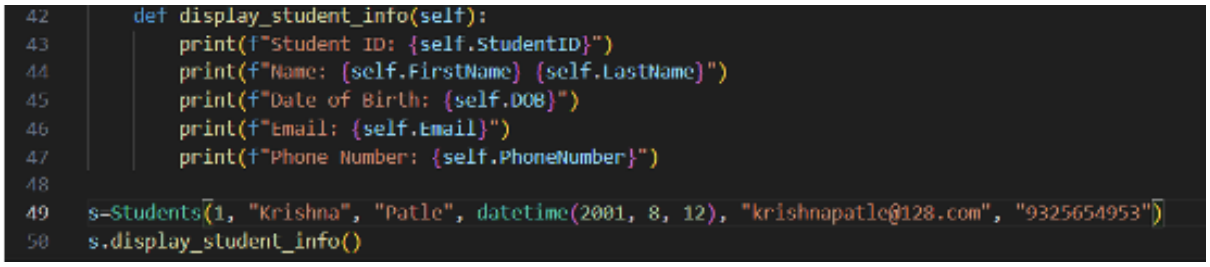
```python
    def update_student_info(self, first_name, last_name, dob, email, phone_number):
        self.FirstName = first_name
        self.LastName = last_name
        self.DateOfBirth = dob
        self.Email = email
        self.PhoneNumber = phone_number
        print("Student information updated successfully.")


s=Students(1, "John", "Doe", datetime(1990, 5, 15), "john.doe@example.com", "123-456-7890")
s.update_student_info("John", "Doe", datetime(1990, 5, 15), "john.doe@example.com", "987-654-3210")
```

• MakePayment(amount: decimal, paymentDate: DateTime): Records a payment made by the student.



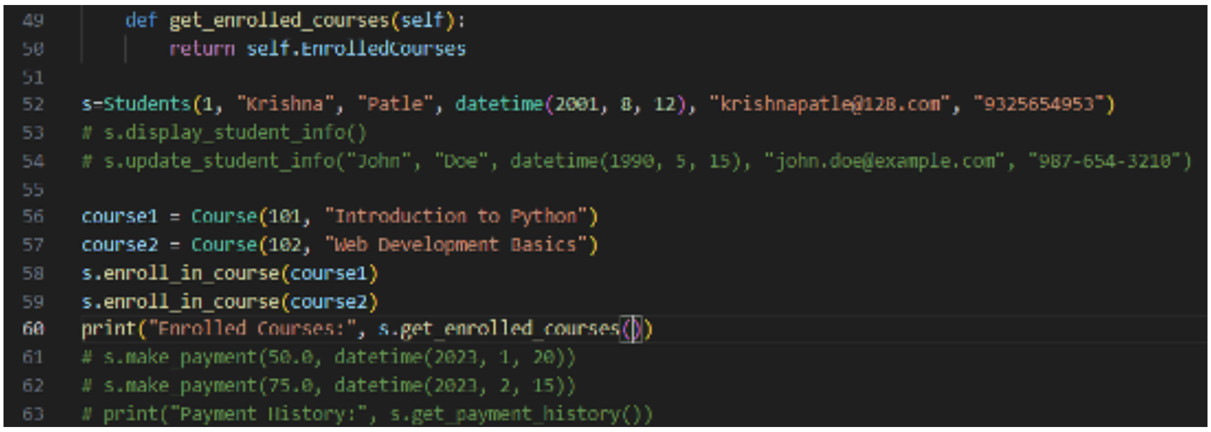
```python
    def make_payment(self, amount, payment_date):
        payment = Payment(amount, payment_date)
        self.PaymentHistory.append(payment)
        print(f"Payment of ${amount} recorded on {payment_date}")

s=Students(1, "John", "Doe", datetime(1990, 5, 15), "john.doe@example.com", "123-456-7890")
# s.update_student_info("John", "Doe", datetime(1990, 5, 15), "john.doe@example.com", "987-654-3210")

# course1 = Course(101, "Introduction to Python")
# course2 = Course(102, "Web Development Basics")
# s.enroll_in_course(course1)
# s.enroll_in_course(course2)

s.make_payment(50.0, datetime(2023, 1, 20))
s.make_payment(75.0, datetime(2023, 2, 15))
```

• DisplayStudentInfo(): Displays detailed information about the student.



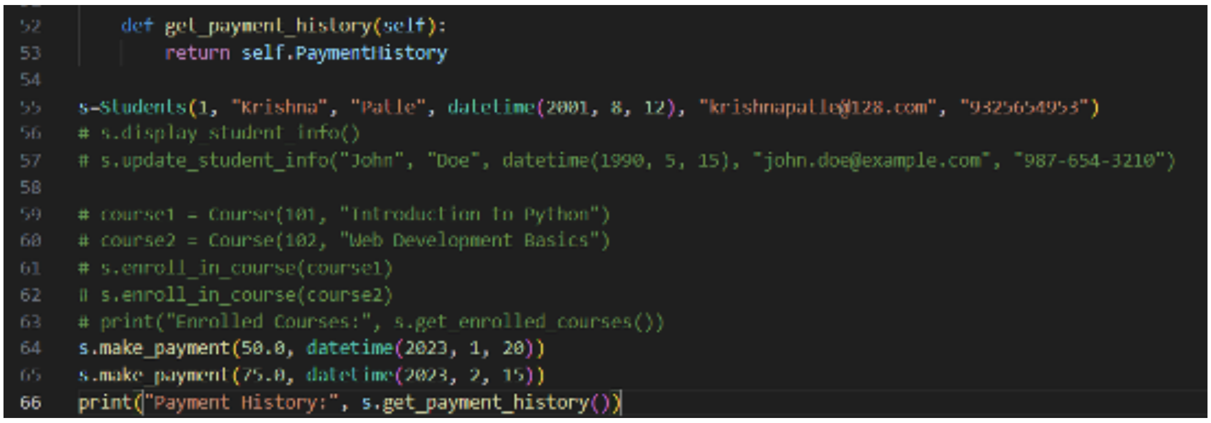
```python
    def display_student_info(self):
        print(f"Student ID: {self.StudentID}")
        print(f"Name: {self.FirstName} {self.LastName}")
        print(f"Date of Birth: {self.DOB}")
        print(f"Email: {self.Email}")
        print(f"Phone Number: {self.PhoneNumber}")

s=Students(1, "Krishna", "Patle", datetime(2001, 8, 12), "krishnapatle@128.com", "9325654953")
s.display_student_info()
```

• GetEnrolledCourses(): Retrieves a list of courses in which the student is enrolled.



```python
    def get_enrolled_courses(self):
        return self.EnrolledCourses

s=Students(1, "Krishna", "Patle", datetime(2001, 8, 12), "krishnapatle@128.com", "9325654953")
# s.display_student_info()
# s.update_student_info("John", "Doe", datetime(1990, 5, 15), "john.doe@example.com", "987-654-3210")

course1 = Course(101, "Introduction to Python")
course2 = Course(102, "Web Development Basics")
s.enroll_in_course(course1)
s.enroll_in_course(course2)
print("Enrolled Courses:", s.get_enrolled_courses())
# s.make_payment(50.0, datetime(2023, 1, 20))
# s.make_payment(75.0, datetime(2023, 2, 15))
# print("Payment History:", s.get_payment_history())
```

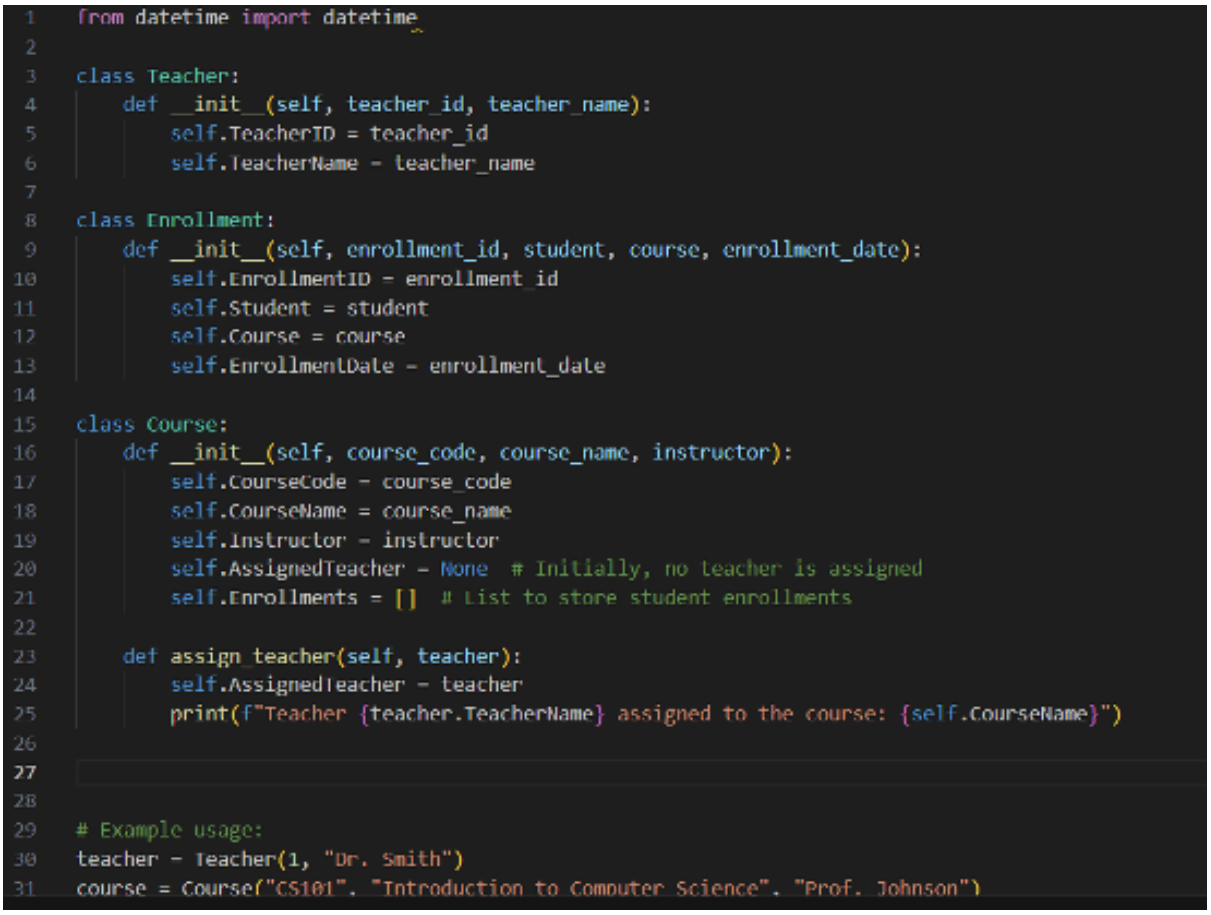
• GetPaymentHistory(): Retrieves a list of payment records for the student.



```python
    def get_payment_history(self):
        return self.PaymentHistory

s=Students(1, "Krishna", "Patle", datetime(2001, 8, 12), "krishnapatle@128.com", "9325654953")
# s.display_student_info()
# s.update_student_info("John", "Doe", datetime(1990, 5, 15), "john.doe@example.com", "987-654-3210")

# course1 = Course(101, "Introduction to Python")
# course2 = Course(102, "Web Development Basics")
# s.enroll_in_course(course1)
# s.enroll_in_course(course2)
# print("Enrolled Courses:", s.get_enrolled_courses())
s.make_payment(50.0, datetime(2023, 1, 20))
s.make_payment(75.0, datetime(2023, 2, 15))
print("Payment History:", s.get_payment_history())
```

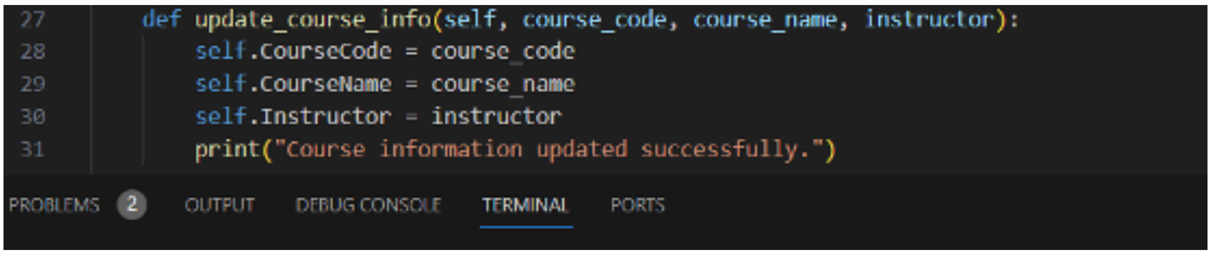
Course Class:

- AssignTeacher(teacher: Teacher): Assigns a teacher to the course.



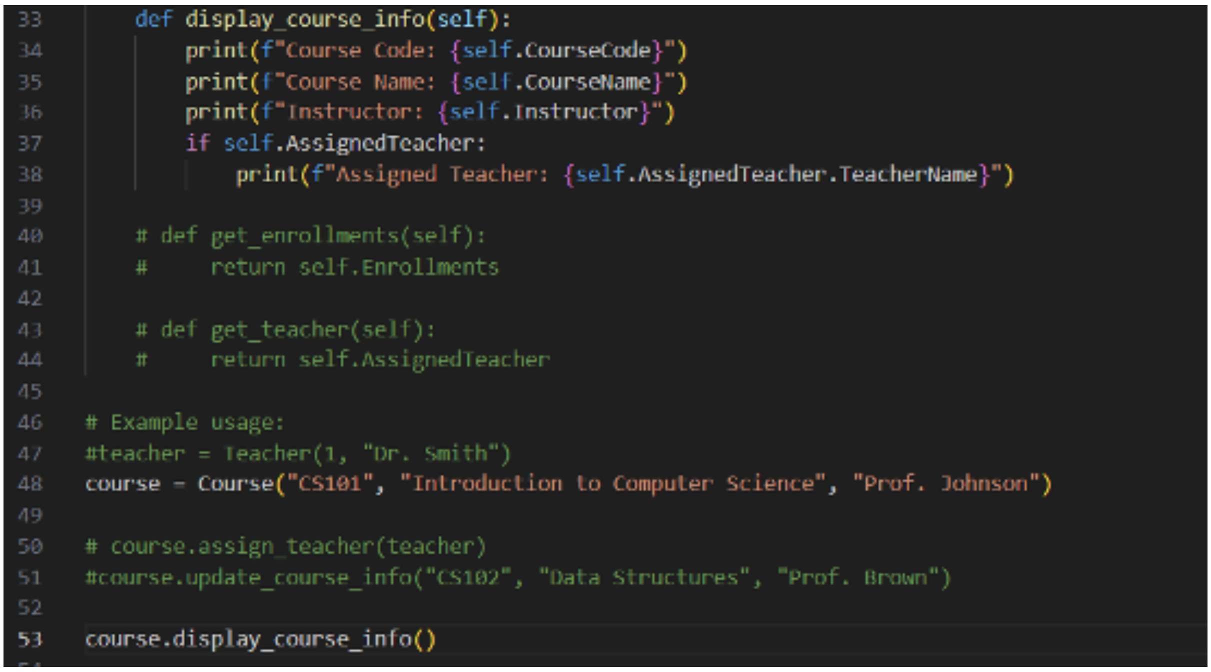
```python
from datetime import datetime

class Teacher:
    def __init__(self, teacher_id, teacher_name):
        self.TeacherID = teacher_id
        self.TeacherName = teacher_name

class Enrollment:
    def __init__(self, enrollment_id, student, course, enrollment_date):
        self.EnrollmentID = enrollment_id
        self.Student = student
        self.Course = course
        self.EnrollmentDate = enrollment_date

class Course:
    def __init__(self, course_code, course_name, instructor):
        self.CourseCode = course_code
        self.CourseName = course_name
        self.Instructor = instructor
        self.AssignedTeacher = None  # Initially, no teacher is assigned
        self.Enrollments = []  # List to store student enrollments

    def assign_teacher(self, teacher):
        self.AssignedTeacher = teacher
        print(f"Teacher {teacher.TeacherName} assigned to the course: {self.CourseName}")



# Example usage:
teacher = Teacher(1, "Dr. Smith")
course = Course("CS101", "Introduction to Computer Science", "Prof. Johnson")
```

- UpdateCourseInfo(courseCode: string, courseName: string, instructor: string): Updates course information.



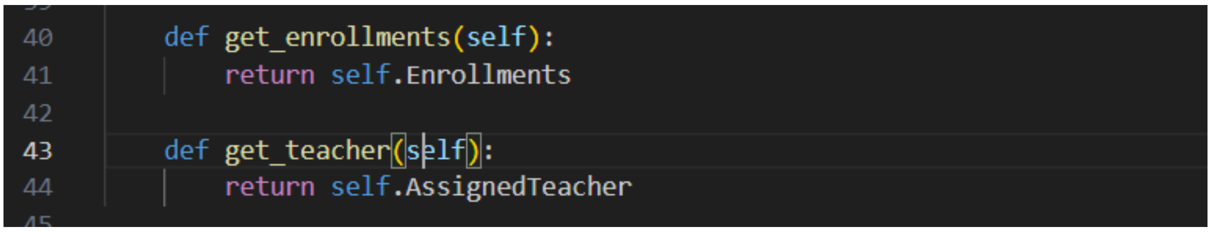
```python
    def update_course_info(self, course_code, course_name, instructor):
        self.CourseCode = course_code
        self.CourseName = course_name
        self.Instructor = instructor
        print("Course information updated successfully.")
```

- DisplayCourseInfo(): Displays detailed information about the course.



```python
    def display_course_info(self):
        print(f"Course Code: {self.CourseCode}")
        print(f"Course Name: {self.CourseName}")
        print(f"Instructor: {self.Instructor}")
        if self.AssignedTeacher:
            print(f"Assigned Teacher: {self.AssignedTeacher.TeacherName}")

    # def get_enrollments(self):
    #     return self.Enrollments

    # def get_teacher(self):
    #     return self.AssignedTeacher

# Example usage:
#teacher = Teacher(1, "Dr. Smith")
course = Course("CS101", "Introduction to Computer Science", "Prof. Johnson")

# course.assign_teacher(teacher)
#course.update_course_info("CS102", "Data Structures", "Prof. Brown")

course.display_course_info()
```

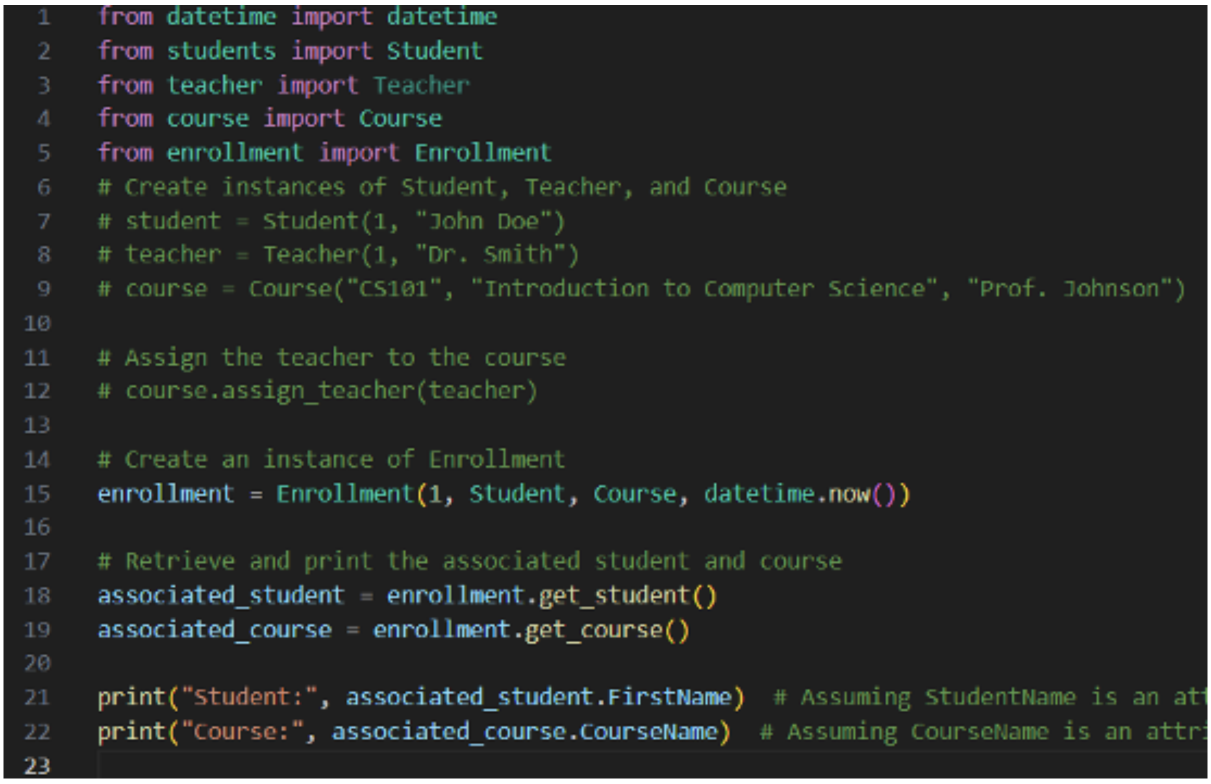
- GetEnrollments(): Retrieves a list of student enrollments for the course.
- GetTeacher(): Retrieves the assigned teacher for the course.



```python
    def get_enrollments(self):
        return self.Enrollments

    def get_teacher(self):
        return self.AssignedTeacher
```

Enrollment Class:

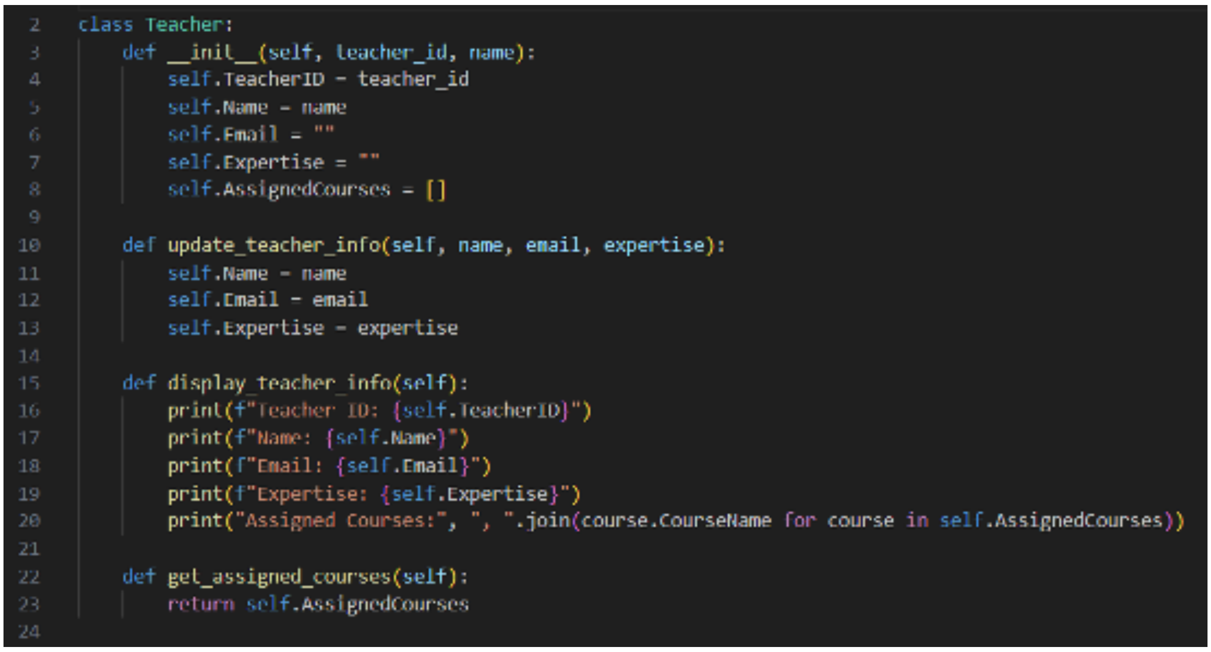
- GetStudent(): Retrieves the student associated with the enrollment.
- GetCourse(): Retrieves the course associated with the enrollment.



```python
from datetime import datetime
from students import Student
from teacher import Teacher
from course import Course
from enrollment import Enrollment
# Create instances of Student, Teacher, and Course
# student = Student(1, "John Doe")
# teacher = Teacher(1, "Dr. Smith")
# course = Course("CS101", "Introduction to Computer Science", "Prof. Johnson")

# Assign the teacher to the course
# course.assign_teacher(teacher)

# Create an instance of Enrollment
enrollment = Enrollment(1, Student, Course, datetime.now())

# Retrieve and print the associated student and course
associated_student = enrollment.get_student()
associated_course = enrollment.get_course()

print("Student:", associated_student.FirstName)  # Assuming StudentName is an att
print("Course:", associated_course.CourseName)  # Assuming CourseName is an attri

```

Teacher Class:

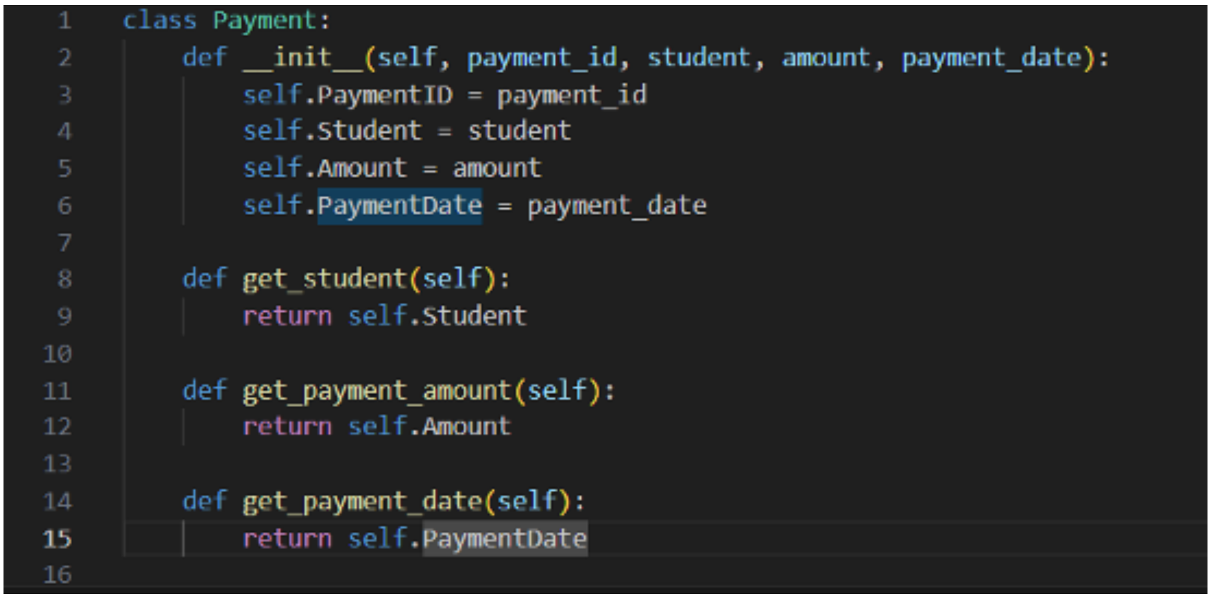
- UpdateTeacherInfo(name: string, email: string, expertise: string): Updates teacher information.
- DisplayTeacherInfo(): Displays detailed information about the teacher
- GetAssignedCourses(): Retrieves a list of courses assigned to the teacher.



```python
class Teacher:
    def __init__(self, teacher_id, name):
        self.TeacherID = teacher_id
        self.Name = name
        self.Email = ""
        self.Expertise = ""
        self.AssignedCourses = []

    def update_teacher_info(self, name, email, expertise):
        self.Name = name
        self.Email = email
        self.Expertise = expertise

    def display_teacher_info(self):
        print(f"Teacher ID: {self.TeacherID}")
        print(f"Name: {self.Name}")
        print(f"Email: {self.Email}")
        print(f"Expertise: {self.Expertise}")
        print("Assigned Courses:", ", ".join(course.CourseName for course in self.AssignedCourses))

    def get_assigned_courses(self):
        return self.AssignedCourses

```

Payment Class:

- GetStudent(): Retrieves the student associated with the payment.
- GetPaymentAmount(): Retrieves the payment amount.
- GetPaymentDate(): Retrieves the payment date.



```python
class Payment:
    def __init__(self, payment_id, student, amount, payment_date):
        self.PaymentID = payment_id
        self.Student = student
        self.Amount = amount
        self.PaymentDate = payment_date

    def get_student(self):
        return self.Student

    def get_payment_amount(self):
        return self.Amount

    def get_payment_date(self):
        return self.PaymentDate

```

SIS Class (if you have one to manage interactions):

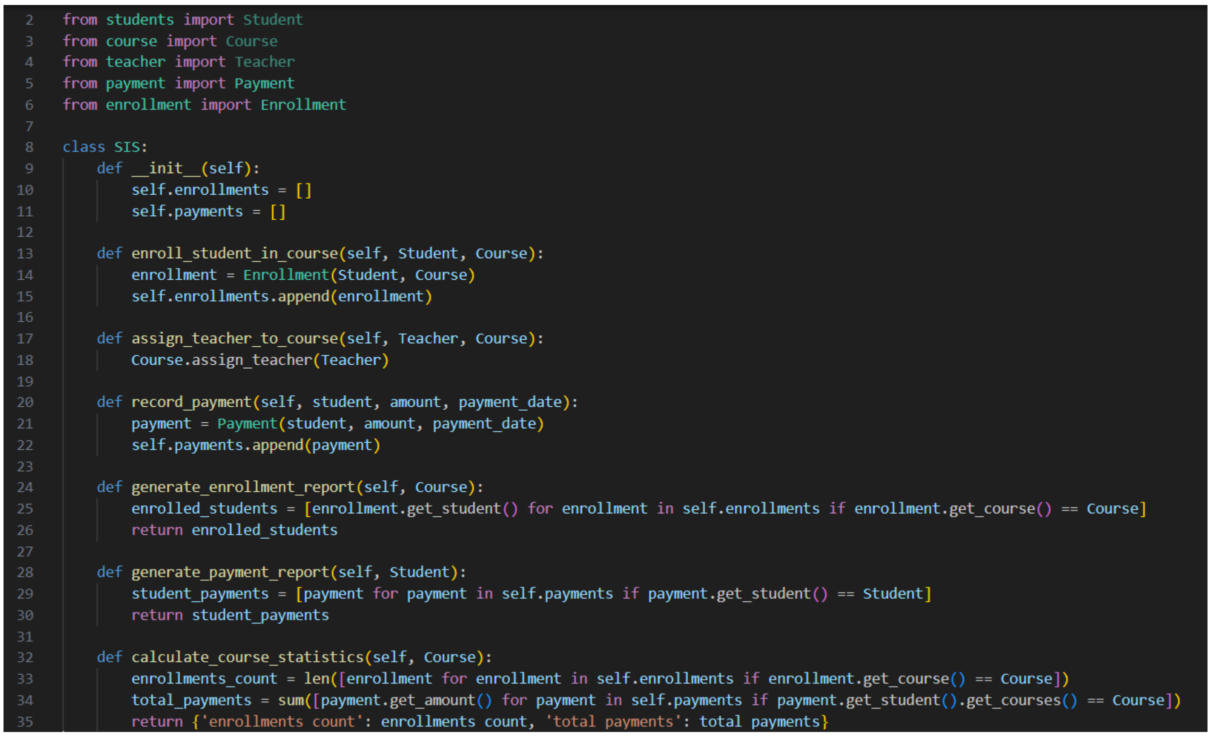
- EnrollStudentInCourse(student: Student, course: Course): Enrolls a student in a course.
- AssignTeacherToCourse(teacher: Teacher, course: Course): Assigns a teacher to a course.
- RecordPayment(student: Student, amount: decimal, paymentDate: DateTime): Records a

  payment made by a student.
- GenerateEnrollmentReport(course: Course): Generates a report of students enrolled in a

  specific course.
- GeneratePaymentReport(student: Student): Generates a report of payments made by a specific

  student.
- CalculateCourseStatistics(course: Course): Calculates statistics for a specific course, such as the

  number of enrollments and total payments.



```python
from students import Student
from course import Course
from teacher import Teacher
from payment import Payment
from enrollment import Enrollment

class SIS:
    def __init__(self):
        self.enrollments = []
        self.payments = []

    def enroll_student_in_course(self, Student, Course):
        enrollment = Enrollment(Student, Course)
        self.enrollments.append(enrollment)

    def assign_teacher_to_course(self, Teacher, Course):
        Course.assign_teacher(Teacher)

    def record_payment(self, student, amount, payment_date):
        payment = Payment(student, amount, payment_date)
        self.payments.append(payment)

    def generate_enrollment_report(self, Course):
        enrolled_students = [enrollment.get_student() for enrollment in self.enrollments if enrollment.get_course() == Course]
        return enrolled_students

    def generate_payment_report(self, Student):
        student_payments = [payment for payment in self.payments if payment.get_student() == Student]
        return student_payments

    def calculate_course_statistics(self, Course):
        enrollments_count = len([enrollment for enrollment in self.enrollments if enrollment.get_course() == Course])
        total_payments = sum([payment.get_amount() for payment in self.payments if payment.get_student().get_courses() == Course])
        return {'enrollments count': enrollments count, 'total payments': total payments}
```

Task 4: Exceptions handling and Custom Exceptions

Implementing custom exceptions allows you to define and throw exceptions tailored to specific situations or business logic requirements.

Create Custom Exception Classes

You'll need to create custom exception classes that are inherited from the System.Exception class or one of its derived classes (e.g., System.ApplicationException). These custom exception classes will allow you to encapsulate specific error scenarios and provide meaningful error messages.

Throw Custom Exceptions In your code, you can throw custom exceptions when specific conditions or business logic rules are violated. To throw a custom exception, use the throw keyword followed by an instance of your custom exception class.

- **DuplicateEnrollmentException:** Thrown when a student is already enrolled in a course and tries to enroll again. This exception can be used in the EnrollStudentInCourse method.

- **CourseNotFoundException:** Thrown when a course does not exist in the system, and you attempt to perform operations on it (e.g., enrolling a student or assigning a teacher).

- **StudentNotFoundException:** Thrown when a student does not exist in the system, and you attempt to perform operations on the student (e.g., enrolling in a course, making a payment).

- TeacherNotFoundException: Thrown when a teacher does not exist in the system, and you attempt to assign them to a course.

• PaymentValidationException: Thrown when there is an issue with payment validation, such as an invalid payment amount or payment date.

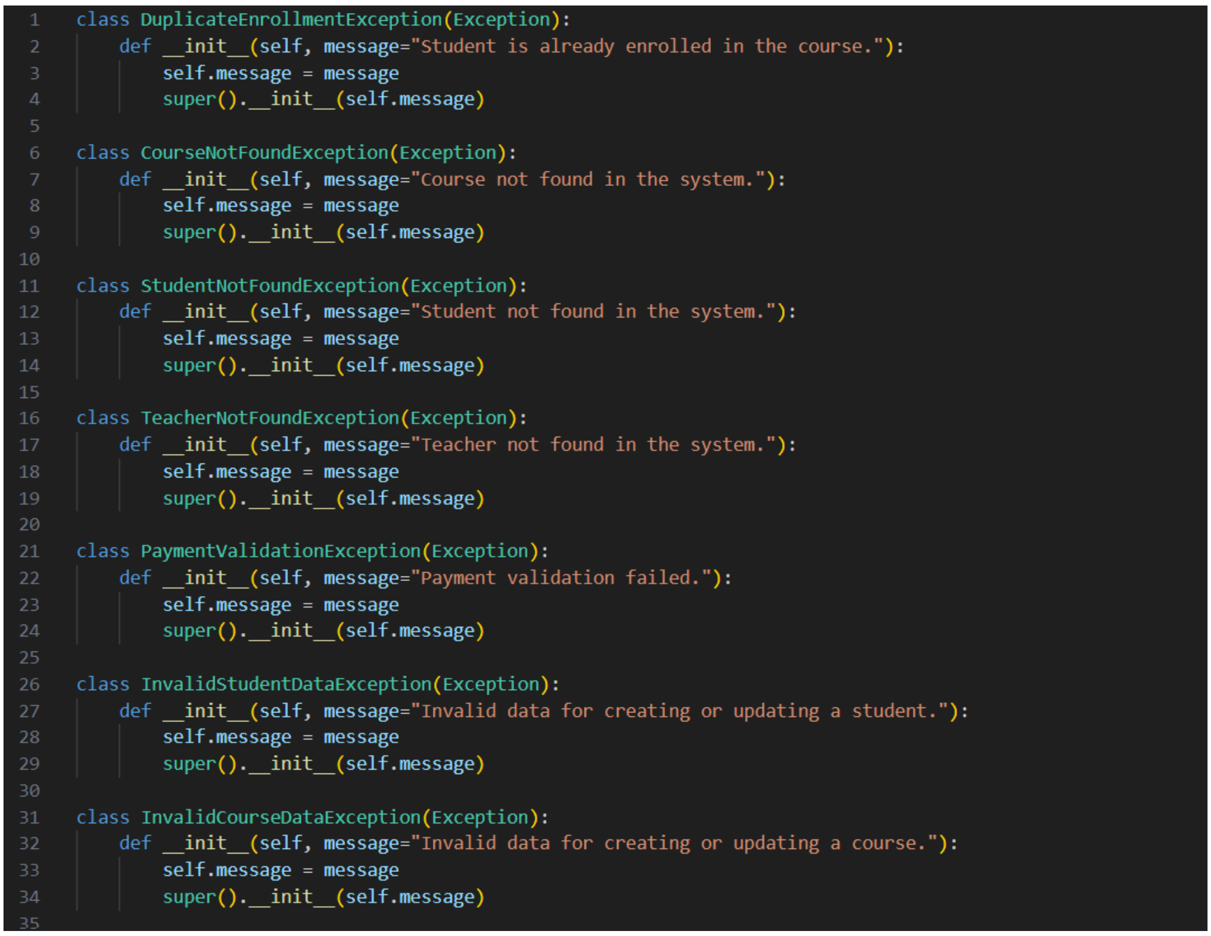
• InvalidStudentDataException: Thrown when data provided for creating or updating a student is invalid (e.g., invalid date of birth or email format).

• InvalidCourseDataException: Thrown when data provided for creating or updating a course is invalid (e.g., invalid course code or instructor name).

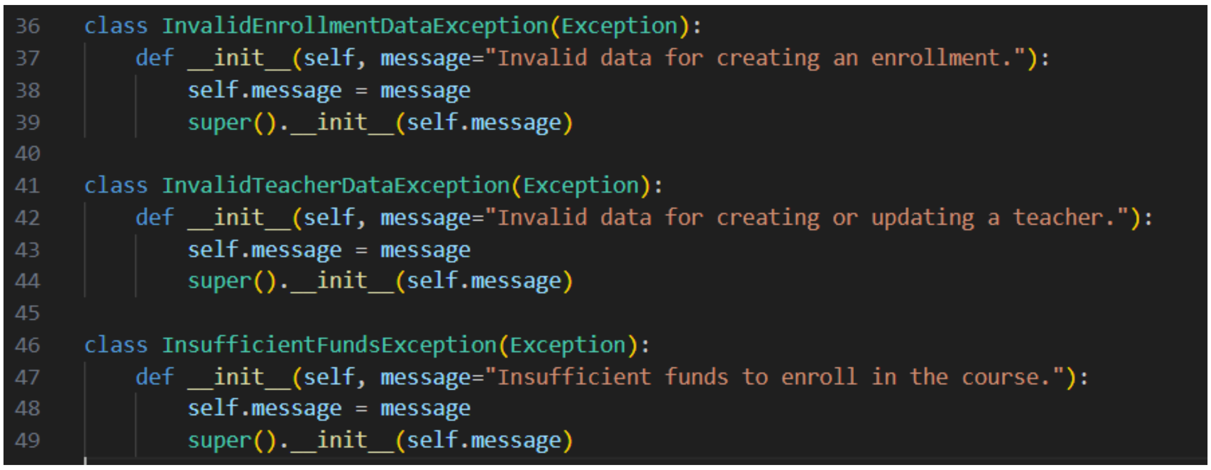
• InvalidEnrollmentDataException: Thrown when data provided for creating an enrollment isinvalid (e.g., missing student or course references).

• InvalidTeacherDataException: Thrown when data provided for creating or updating a teacher is invalid (e.g., missing name or email).

• InsufficientFundsException: Thrown when a student attempts to enroll in a course but does not have enough funds to make the payment



```python
class DuplicateEnrollmentException(Exception):
    def __init__(self, message="Student is already enrolled in the course."):
        self.message = message
        super().__init__(self.message)

class CourseNotFoundException(Exception):
    def __init__(self, message="Course not found in the system."):
        self.message = message
        super().__init__(self.message)

class StudentNotFoundException(Exception):
    def __init__(self, message="Student not found in the system."):
        self.message = message
        super().__init__(self.message)

class TeacherNotFoundException(Exception):
    def __init__(self, message="Teacher not found in the system."):
        self.message = message
        super().__init__(self.message)

class PaymentValidationException(Exception):
    def __init__(self, message="Payment validation failed."):
        self.message = message
        super().__init__(self.message)

class InvalidStudentDataException(Exception):
    def __init__(self, message="Invalid data for creating or updating a student."):
        self.message = message
        super().__init__(self.message)

class InvalidCourseDataException(Exception):
    def __init__(self, message="Invalid data for creating or updating a course."):
        self.message = message
        super().__init__(self.message)

```


```python
class InvalidEnrollmentDataException(Exception):
    def __init__(self, message="Invalid data for creating an enrollment."):
        self.message = message
        super().__init__(self.message)

class InvalidTeacherDataException(Exception):
    def __init__(self, message="Invalid data for creating or updating a teacher."):
        self.message = message
        super().__init__(self.message)

class InsufficientFundsException(Exception):
    def __init__(self, message="Insufficient funds to enroll in the course."):
        self.message = message
        super().__init__(self.message)
```

Task 6: Create Methods for Managing Relationships

To add, remove, or retrieve related objects, you should create methods within your SIS class or each relevant class.

• **AddEnrollment(student, course, enrollmentDate**): In the SIS class, create a method that adds an enrollment to both the Student's and Course's enrollment lists. Ensure the Enrollment object

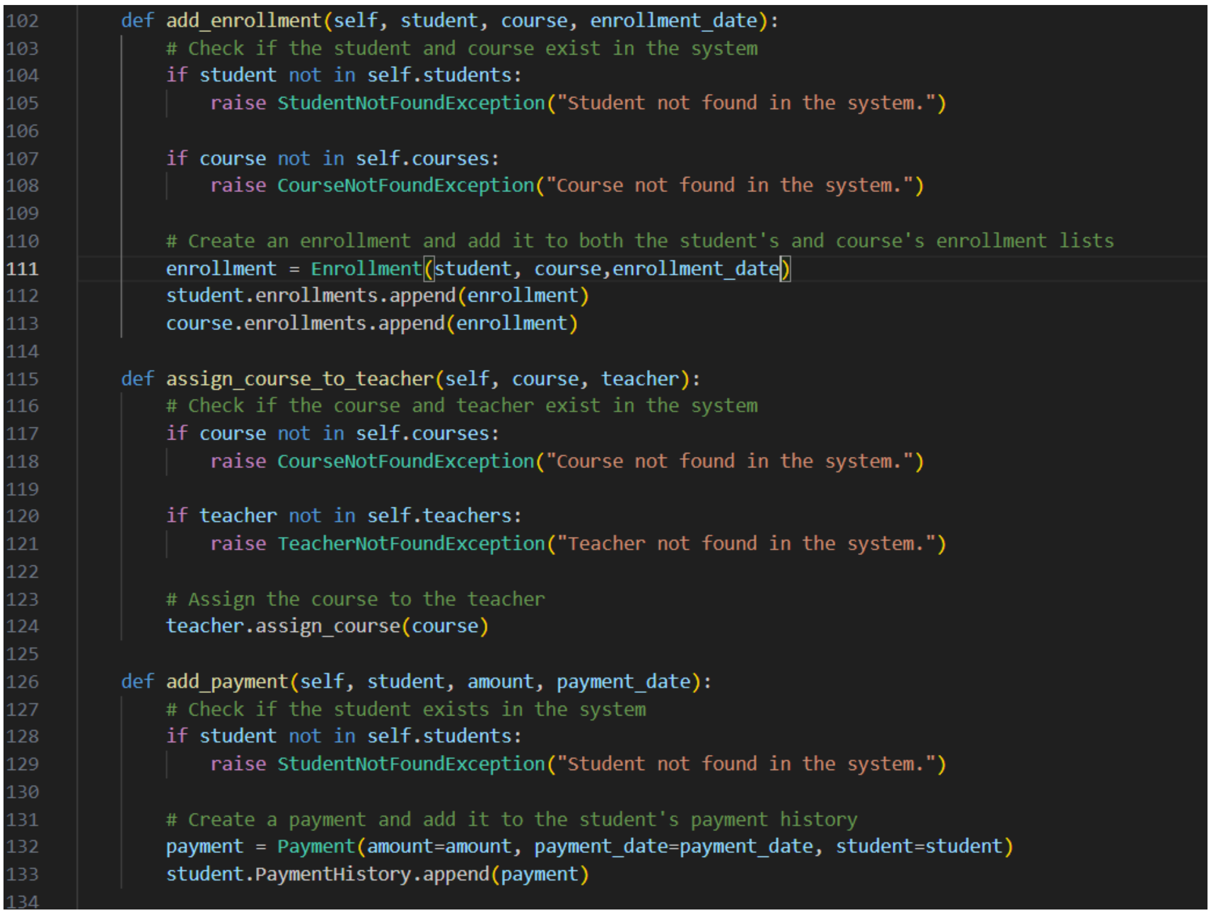
references the correct Student and Course.

• **AssignCourseToTeacher(course, teacher**): In the SIS class, create a method to assign a course to a teacher. Add the course to the teacher's AssignedCourses list.

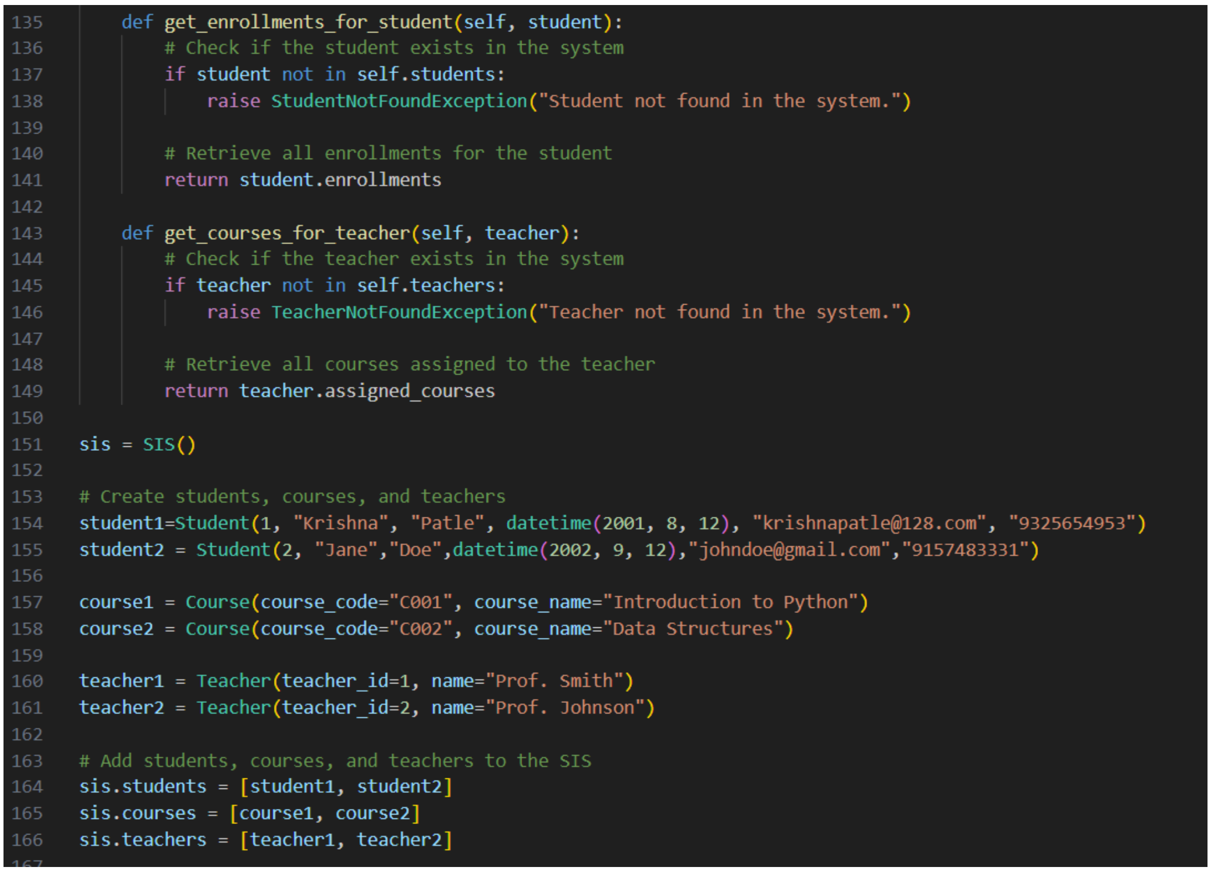
• **AddPayment(student, amount, paymentDate**): In the SIS class, create a method that adds a payment to the Student's payment history. Ensure the Payment object references the correct Student.

• **GetEnrollmentsForStudent(student**): In the SIS class, create a method to retrieve all enrollments for a specific student.

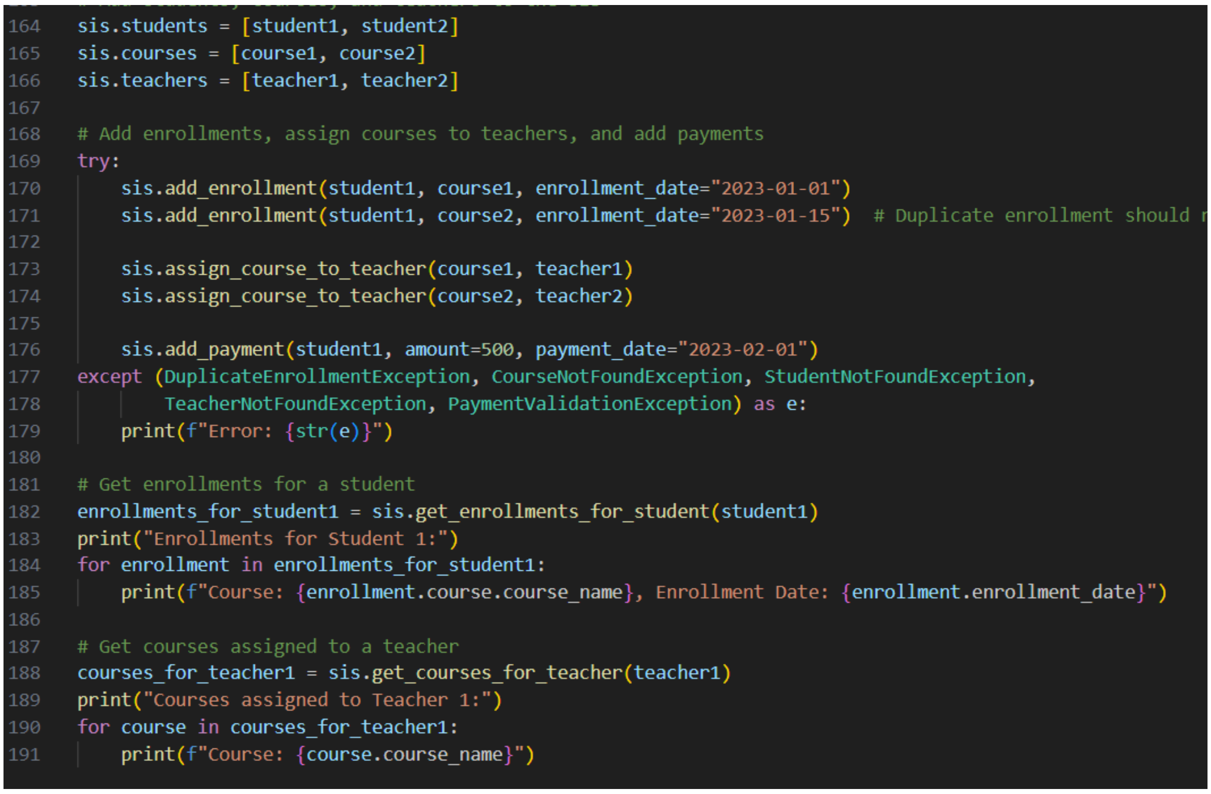
• **GetCoursesForTeacher(teacher**): In the SIS class, create a method to retrieve all courses assigned to a specific teacher.



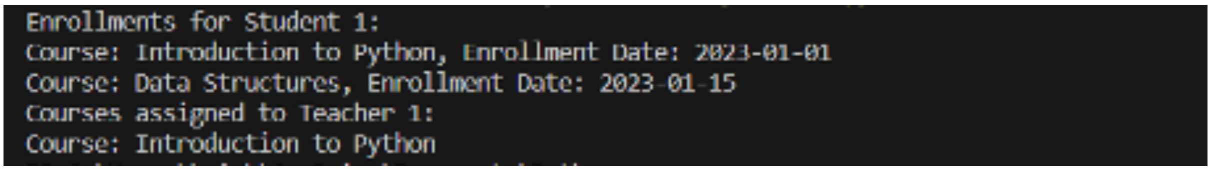
```python
    def add_enrollment(self, student, course, enrollment_date):
        # Check if the student and course exist in the system
        if student not in self.students:
            raise StudentNotFoundException("Student not found in the system.")

        if course not in self.courses:
            raise CourseNotFoundException("Course not found in the system.")

        # Create an enrollment and add it to both the student's and course's enrollment lists
        enrollment = Enrollment(student, course,enrollment_date)
        student.enrollments.append(enrollment)
        course.enrollments.append(enrollment)

    def assign_course_to_teacher(self, course, teacher):
        # Check if the course and teacher exist in the system
        if course not in self.courses:
            raise CourseNotFoundException("Course not found in the system.")

        if teacher not in self.teachers:
            raise TeacherNotFoundException("Teacher not found in the system.")

        # Assign the course to the teacher
        teacher.assign_course(course)

    def add_payment(self, student, amount, payment_date):
        # Check if the student exists in the system
        if student not in self.students:
            raise StudentNotFoundException("Student not found in the system.")

        # Create a payment and add it to the student's payment history
        payment = Payment(amount=amount, payment_date=payment_date, student=student)
        student.PaymentHistory.append(payment)

```



```python
    def get_enrollments_for_student(self, student):
        # Check if the student exists in the system
        if student not in self.students:
            raise StudentNotFoundException("Student not found in the system.")

        # Retrieve all enrollments for the student
        return student.enrollments

    def get_courses_for_teacher(self, teacher):
        # Check if the teacher exists in the system
        if teacher not in self.teachers:
            raise TeacherNotFoundException("Teacher not found in the system.")

        # Retrieve all courses assigned to the teacher
        return teacher.assigned_courses

sis = SIS()

# Create students, courses, and teachers
student1=Student(1, "Krishna", "Patle", datetime(2001, 8, 12), "krishnapatle@128.com", "9325654953")
student2 = Student(2, "Jane","Doe",datetime(2002, 9, 12),"johndoe@gmail.com","9157483331")

course1 = Course(course_code="C001", course_name="Introduction to Python")
course2 = Course(course_code="C002", course_name="Data Structures")

teacher1 = Teacher(teacher_id=1, name="Prof. Smith")
teacher2 = Teacher(teacher_id=2, name="Prof. Johnson")

# Add students, courses, and teachers to the SIS
sis.students = [student1, student2]
sis.courses = [course1, course2]
sis.teachers = [teacher1, teacher2]
```



```python
sis.students = [student1, student2]
sis.courses = [course1, course2]
sis.teachers = [teacher1, teacher2]

# Add enrollments, assign courses to teachers, and add payments
try:
    sis.add_enrollment(student1, course1, enrollment_date="2023-01-01")
    sis.add_enrollment(student1, course2, enrollment_date="2023-01-15")  # Duplicate enrollment should r

    sis.assign_course_to_teacher(course1, teacher1)
    sis.assign_course_to_teacher(course2, teacher2)

    sis.add_payment(student1, amount=500, payment_date="2023-02-01")
except (DuplicateEnrollmentException, CourseNotFoundException, StudentNotFoundException,
        TeacherNotFoundException, PaymentValidationException) as e:
    print(f"Error: {str(e)}")

# Get enrollments for a student
enrollments_for_student1 = sis.get_enrollments_for_student(student1)
print("Enrollments for Student 1:")
for enrollment in enrollments_for_student1:
    print(f"Course: {enrollment.course.course_name}, Enrollment Date: {enrollment.enrollment_date}")

# Get courses assigned to a teacher
courses_for_teacher1 = sis.get_courses_for_teacher(teacher1)
print("Courses assigned to Teacher 1:")
for course in courses_for_teacher1:
    print(f"Course: {course.course_name}")
```



```
Enrollments for Student 1:
Course: Introduction to Python, Enrollment Date: 2023-01-01
Course: Data Structures, Enrollment Date: 2023-01-15
Courses assigned to Teacher 1:
Course: Introduction to Python
```

Task 7: Database Connectivity

Database Initialization:

Implement a method that initializes a database connection and creates tables for storing student,course, enrollment, teacher, and payment information. Create SQL scripts or use code-first migration to create tables with appropriate schemas for your SIS.

Data Retrieval:

Implement methods to retrieve data from the database. Users should be able to request information about students, courses, enrollments, teachers, or payments. Ensure that the data retrieval methods handle exceptions and edge cases gracefully.

Data Insertion and Updating:

Implement methods to insert new data (e.g., enrollments, payments) into the database and update existing data (e.g., student information). Use methods to perform data insertion and updating.

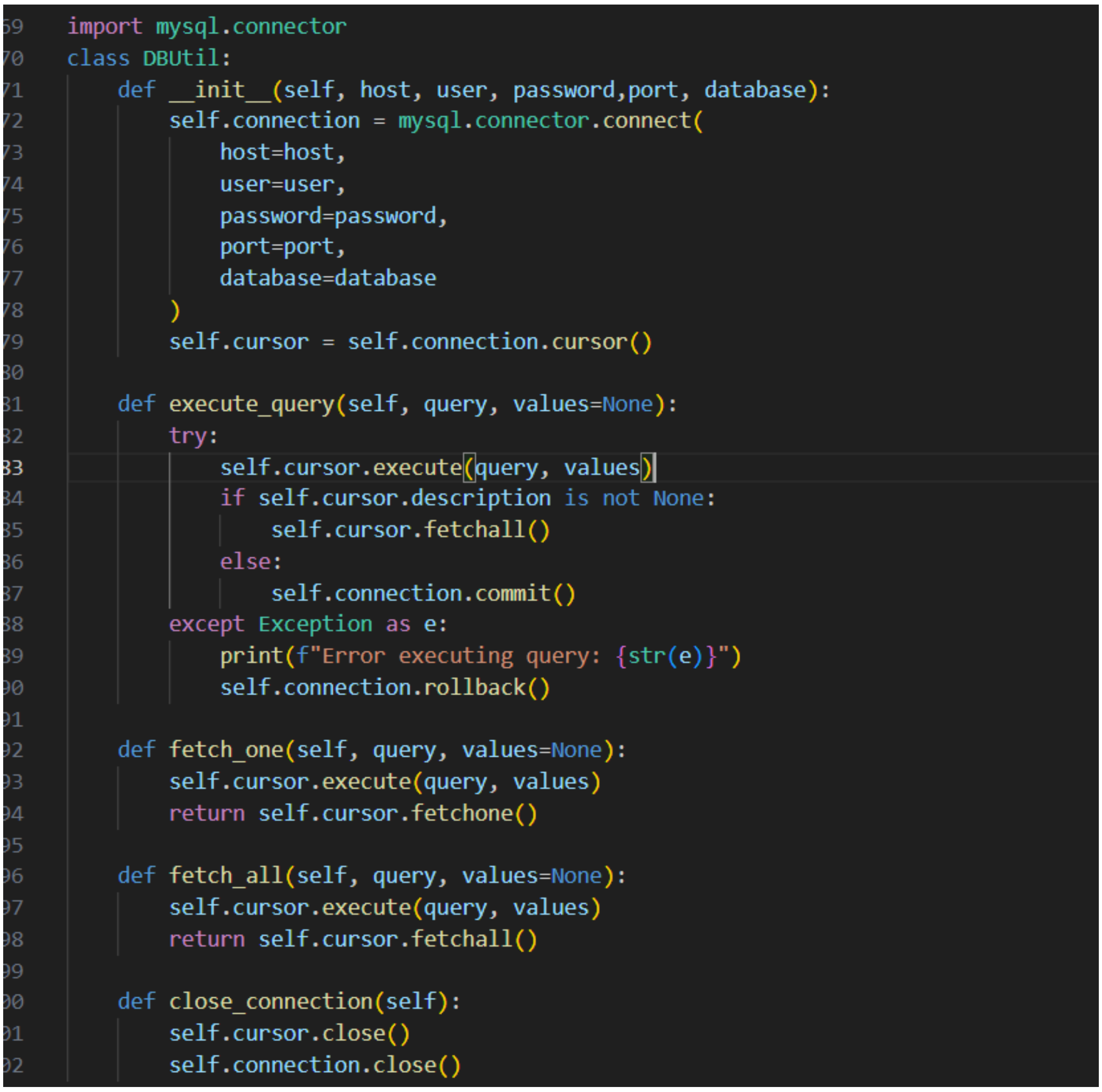
Implement validation checks to ensure data integrity and handle any errors during these operations.

Transaction Management:

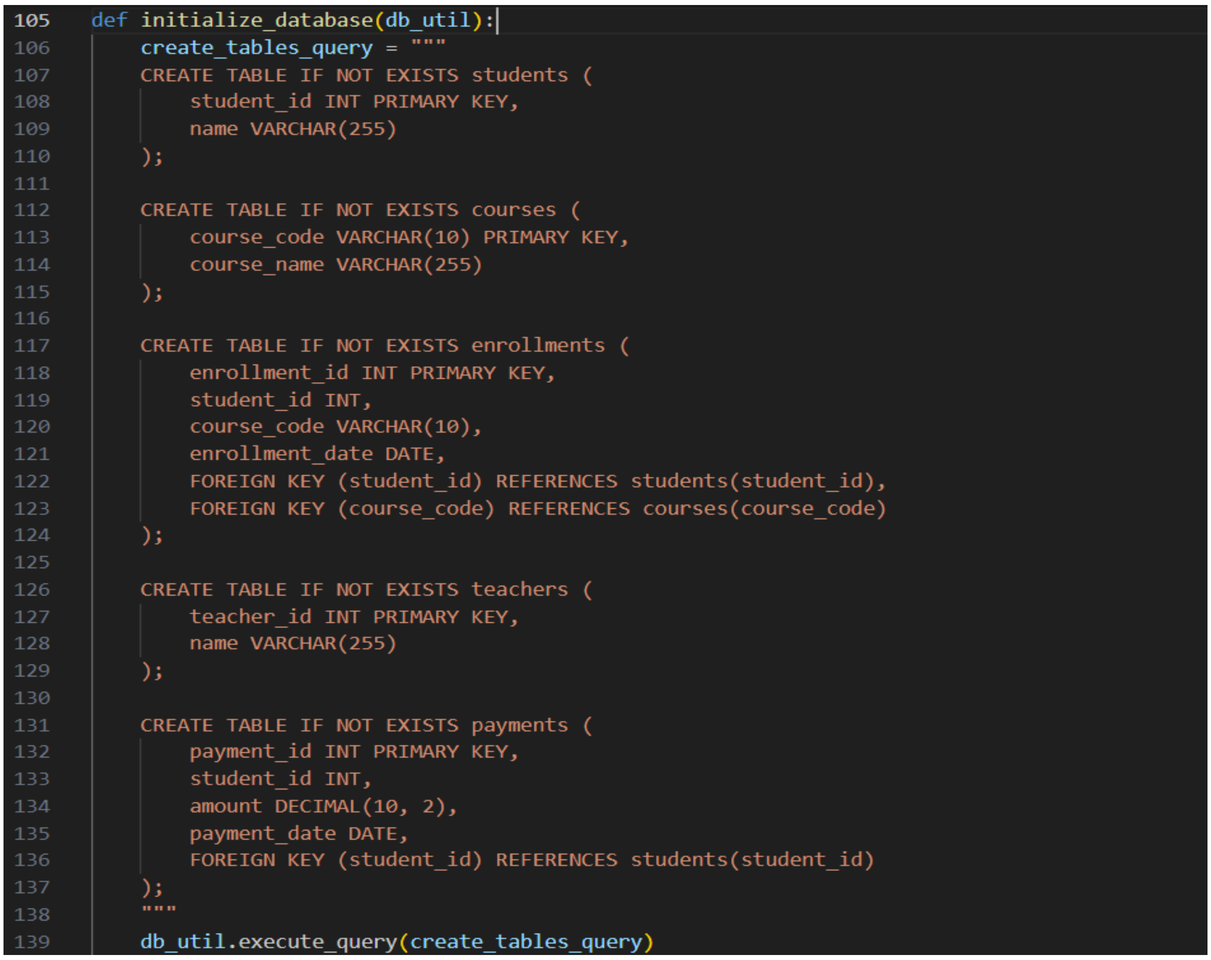
Implement methods for handling database transactions when enrolling students, assigning teachers, or recording payments. Transactions should be atomic and maintain data integrity. Use database transactions to ensure that multiple related operations either all succeed or all fail. Implement error handling and rollback mechanisms in case of transaction failures.

Dynamic Query Builder:

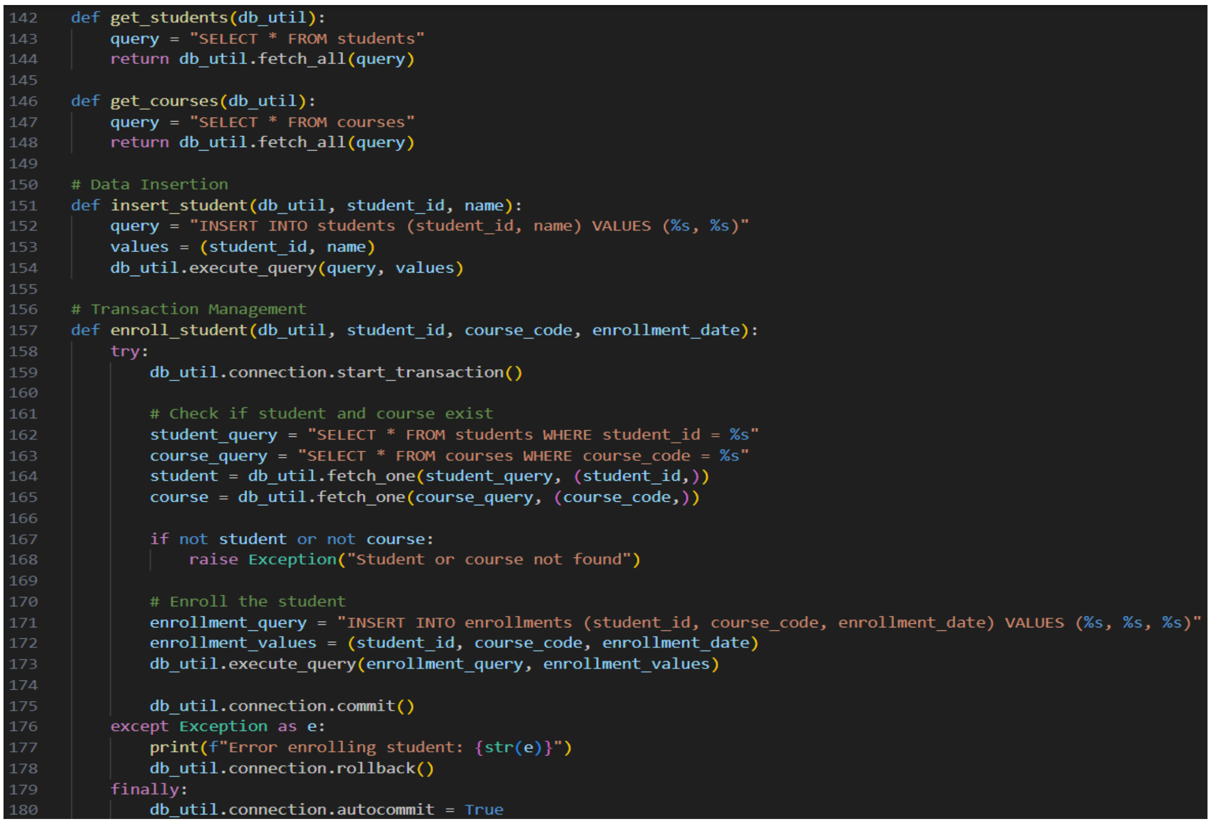
Implement a dynamic query builder that allows users to construct and execute custom SQL queries to retrieve specific data from the database. Users should be able to specify columns, conditions, and sorting criteria. Create a query builder method that dynamically generates SQL queries based on user input. Implement parameterization and sanitation of user inputs to prevent SQL injection.



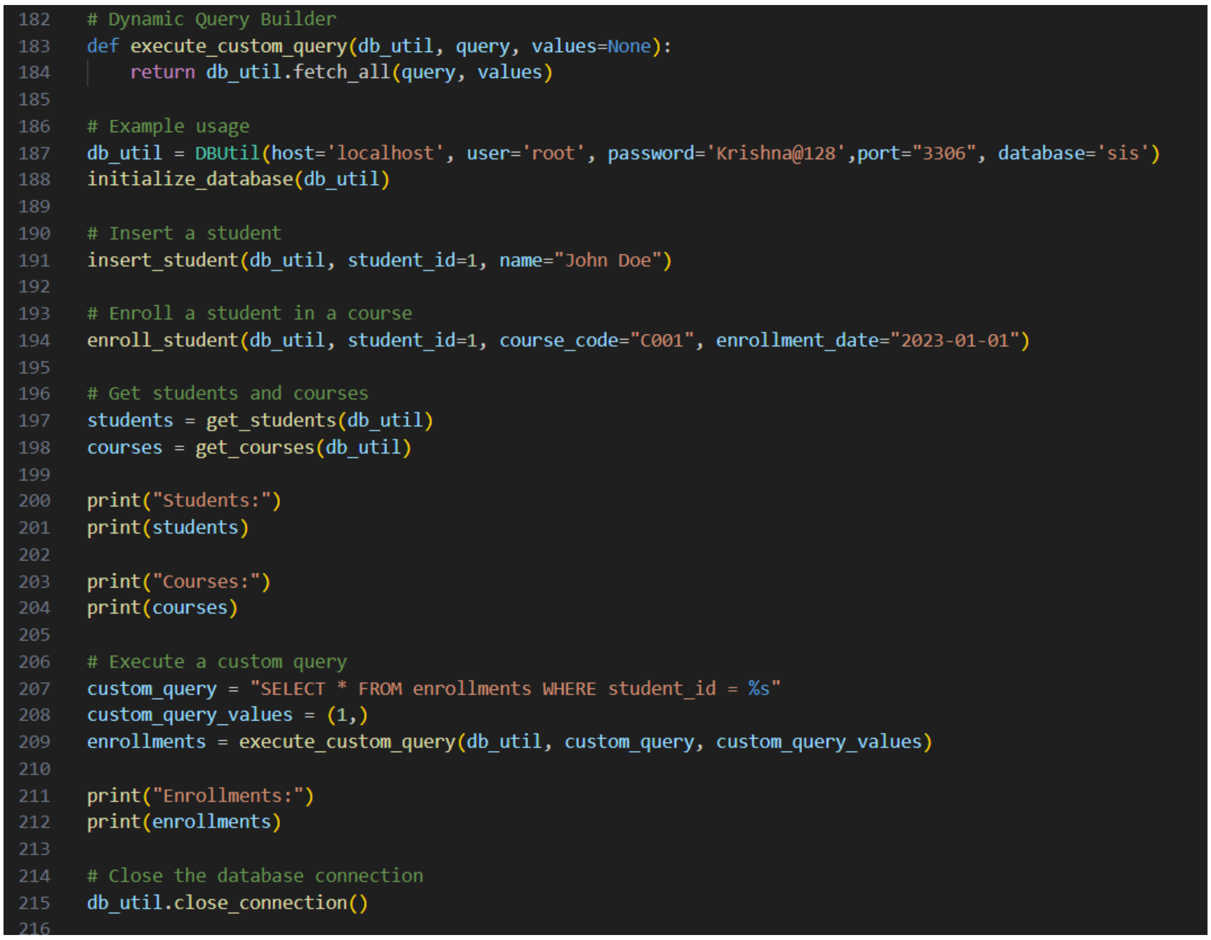
```python
import mysql.connector
class DBUtil:
    def __init__(self, host, user, password,port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def execute_query(self, query, values=None):
        try:
            self.cursor.execute(query, values)
            if self.cursor.description is not None:
                self.cursor.fetchall()
            else:
                self.connection.commit()
        except Exception as e:
            print(f"Error executing query: {str(e)}")
            self.connection.rollback()

    def fetch_one(self, query, values=None):
        self.cursor.execute(query, values)
        return self.cursor.fetchone()

    def fetch_all(self, query, values=None):
        self.cursor.execute(query, values)
        return self.cursor.fetchall()

    def close_connection(self):
        self.cursor.close()
        self.connection.close()
```



```python
def initialize_database(db_util):
    create_tables_query = """
    CREATE TABLE IF NOT EXISTS students (
        student_id INT PRIMARY KEY,
        name VARCHAR(255)
    );

    CREATE TABLE IF NOT EXISTS courses (
        course_code VARCHAR(10) PRIMARY KEY,
        course_name VARCHAR(255)
    );

    CREATE TABLE IF NOT EXISTS enrollments (
        enrollment_id INT PRIMARY KEY,
        student_id INT,
        course_code VARCHAR(10),
        enrollment_date DATE,
        FOREIGN KEY (student_id) REFERENCES students(student_id),
        FOREIGN KEY (course_code) REFERENCES courses(course_code)
    );

    CREATE TABLE IF NOT EXISTS teachers (
        teacher_id INT PRIMARY KEY,
        name VARCHAR(255)
    );

    CREATE TABLE IF NOT EXISTS payments (
        payment_id INT PRIMARY KEY,
        student_id INT,
        amount DECIMAL(10, 2),
        payment_date DATE,
        FOREIGN KEY (student_id) REFERENCES students(student_id)
    );
    """
    db_util.execute_query(create_tables_query)
```



```python
def get_students(db_util):
    query = "SELECT * FROM students"
    return db_util.fetch_all(query)

def get_courses(db_util):
    query = "SELECT * FROM courses"
    return db_util.fetch_all(query)

# Data Insertion
def insert_student(db_util, student_id, name):
    query = "INSERT INTO students (student_id, name) VALUES (%s, %s)"
    values = (student_id, name)
    db_util.execute_query(query, values)

# Transaction Management
def enroll_student(db_util, student_id, course_code, enrollment_date):
    try:
        db_util.connection.start_transaction()

        # Check if student and course exist
        student_query = "SELECT * FROM students WHERE student_id = %s"
        course_query = "SELECT * FROM courses WHERE course_code = %s"
        student = db_util.fetch_one(student_query, (student_id,))
        course = db_util.fetch_one(course_query, (course_code,))

        if not student or not course:
            raise Exception("Student or course not found")

        # Enroll the student
        enrollment_query = "INSERT INTO enrollments (student_id, course_code, enrollment_date) VALUES (%s, %s, %s)"
        enrollment_values = (student_id, course_code, enrollment_date)
        db_util.execute_query(enrollment_query, enrollment_values)

        db_util.connection.commit()
    except Exception as e:
        print(f"Error enrolling student: {str(e)}")
        db_util.connection.rollback()
    finally:
        db_util.connection.autocommit = True
```



```python
# Dynamic Query Builder
def execute_custom_query(db_util, query, values=None):
    return db_util.fetch_all(query, values)

# Example usage
db_util = DBUtil(host='localhost', user='root', password='Krishna@128',port="3306", database='sis')
initialize_database(db_util)

# Insert a student
insert_student(db_util, student_id=1, name="John Doe")

# Enroll a student in a course
enroll_student(db_util, student_id=1, course_code="C001", enrollment_date="2023-01-01")

# Get students and courses
students = get_students(db_util)
courses = get_courses(db_util)

print("Students:")
print(students)

print("Courses:")
print(courses)

# Execute a custom query
custom_query = "SELECT * FROM enrollments WHERE student_id = %s"
custom_query_values = (1,)
enrollments = execute_custom_query(db_util, custom_query, custom_query_values)

print("Enrollments:")
print(enrollments)

# Close the database connection
db_util.close_connection()
```

Task 8: Student Enrollment

In this task, a new student, John Doe, is enrolling in the SIS. The system needs to record John's information, including his personal details, and enroll him in a few courses. Database connectivity is required to store this information.

John Doe's details:

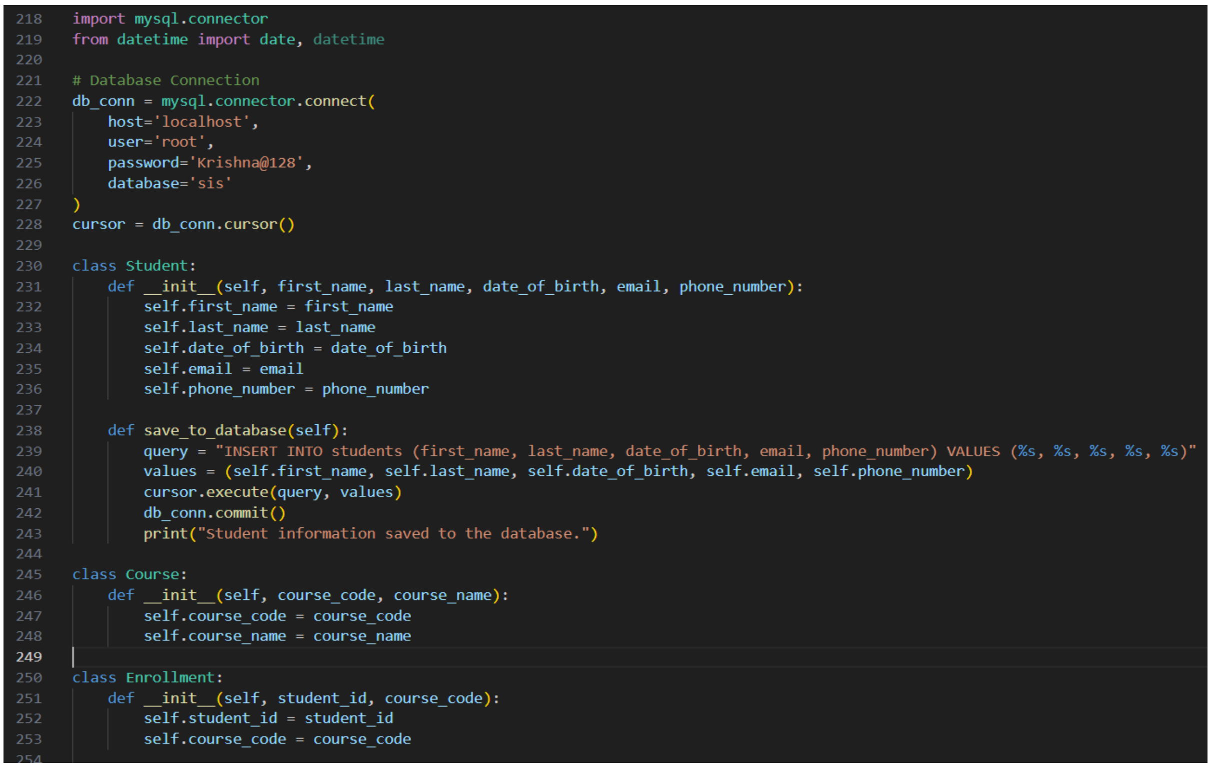
- First Name: John
- Last Name: Doe
- Date of Birth: 1995-08-15
- Email: john.doe@example.com
- Phone Number: 123-456-7890

John is enrolling in the following courses:

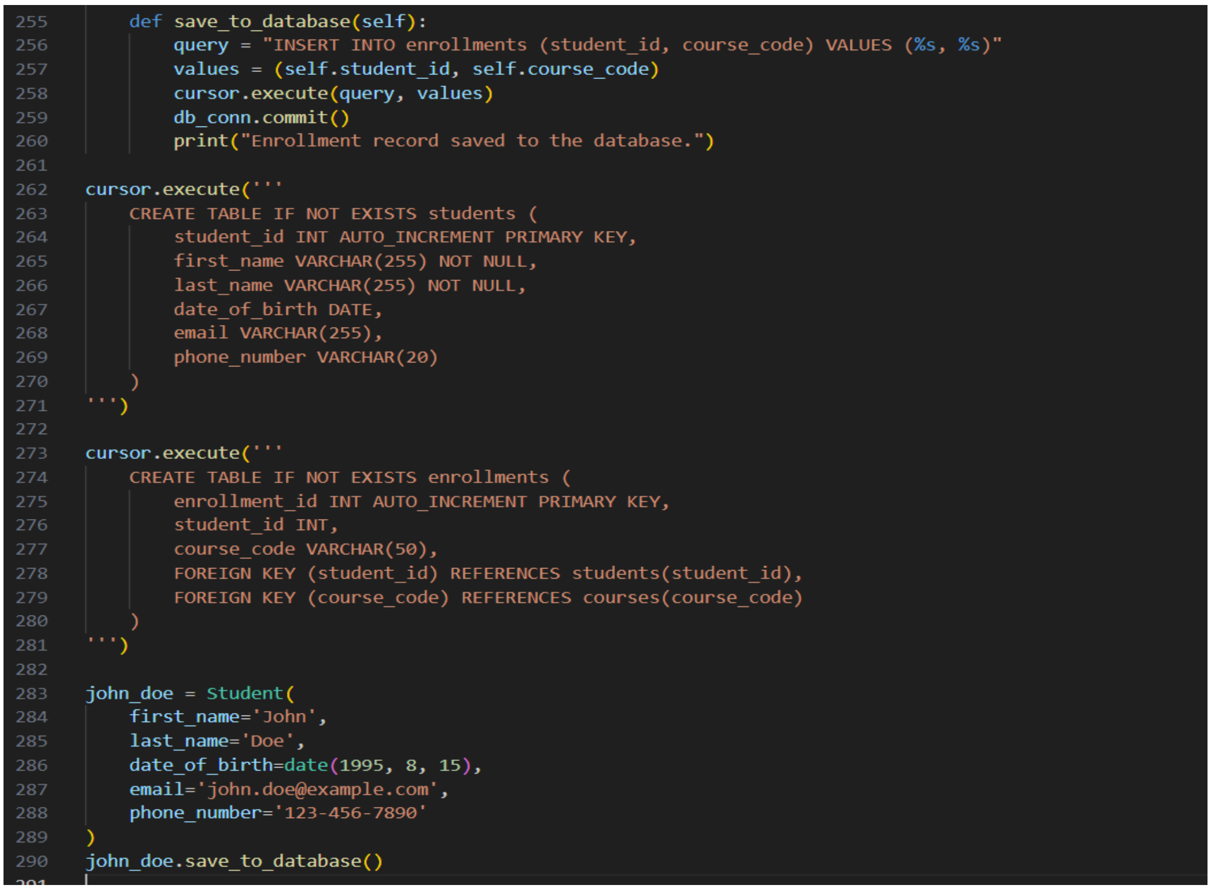
- Course 1: Introduction to Programming
- Course 2: Mathematics 101

The system should perform the following tasks:

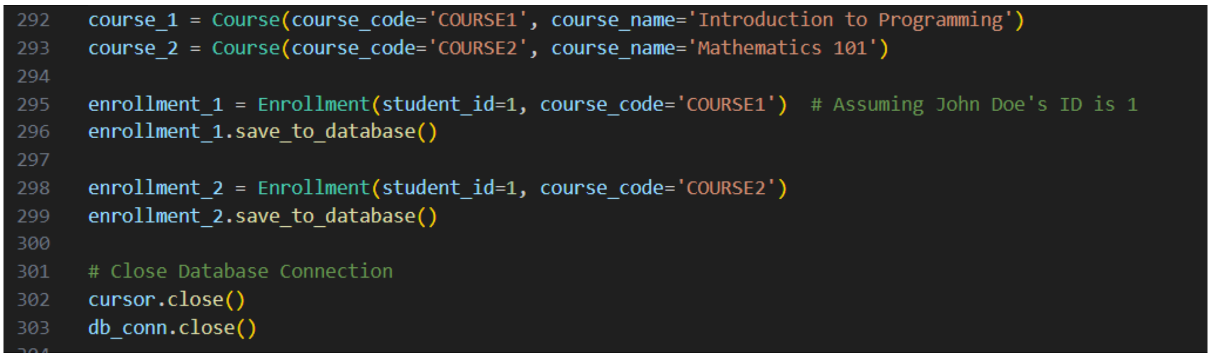
- Create a new student record in the database.
- Enroll John in the specified courses by creating enrollment records in the database.



```python
import mysql.connector
from datetime import date, datetime

# Database Connection
db_conn = mysql.connector.connect(
    host='localhost',
    user='root',
    password='Krishna@128',
    database='sis'
)
cursor = db_conn.cursor()

class Student:
    def __init__(self, first_name, last_name, date_of_birth, email, phone_number):
        self.first_name = first_name
        self.last_name = last_name
        self.date_of_birth = date_of_birth
        self.email = email
        self.phone_number = phone_number

    def save_to_database(self):
        query = "INSERT INTO students (first_name, last_name, date_of_birth, email, phone_number) VALUES (%s, %s, %s, %s, %s)"
        values = (self.first_name, self.last_name, self.date_of_birth, self.email, self.phone_number)
        cursor.execute(query, values)
        db_conn.commit()
        print("Student information saved to the database.")

class Course:
    def __init__(self, course_code, course_name):
        self.course_code = course_code
        self.course_name = course_name

class Enrollment:
    def __init__(self, student_id, course_code):
        self.student_id = student_id
        self.course_code = course_code

```



```python
    def save_to_database(self):
        query = "INSERT INTO enrollments (student_id, course_code) VALUES (%s, %s)"
        values = (self.student_id, self.course_code)
        cursor.execute(query, values)
        db_conn.commit()
        print("Enrollment record saved to the database.")

cursor.execute('''
    CREATE TABLE IF NOT EXISTS students (
        student_id INT AUTO_INCREMENT PRIMARY KEY,
        first_name VARCHAR(255) NOT NULL,
        last_name VARCHAR(255) NOT NULL,
        date_of_birth DATE,
        email VARCHAR(255),
        phone_number VARCHAR(20)
    )
''')

cursor.execute('''
    CREATE TABLE IF NOT EXISTS enrollments (
        enrollment_id INT AUTO_INCREMENT PRIMARY KEY,
        student_id INT,
        course_code VARCHAR(50),
        FOREIGN KEY (student_id) REFERENCES students(student_id),
        FOREIGN KEY (course_code) REFERENCES courses(course_code)
    )
''')

john_doe = Student(
    first_name='John',
    last_name='Doe',
    date_of_birth=date(1995, 8, 15),
    email='john.doe@example.com',
    phone_number='123-456-7890'
)
john_doe.save_to_database()
```


```
course_1 = Course(course_code='COURSE1', course_name='Introduction to Programming')
course_2 = Course(course_code='COURSE2', course_name='Mathematics 101')

enrollment_1 = Enrollment(student_id=1, course_code='COURSE1')  # Assuming John Doe's ID is 1
enrollment_1.save_to_database()

enrollment_2 = Enrollment(student_id=1, course_code='COURSE2')
enrollment_2.save_to_database()

# Close Database Connection
cursor.close()
db_conn.close()
```

Task 9: Teacher Assignment

In this task, a new teacher, Sarah Smith, is assigned to teach a course. The system needs to update the course record to reflect the teacher assignment.

Teacher's Details:

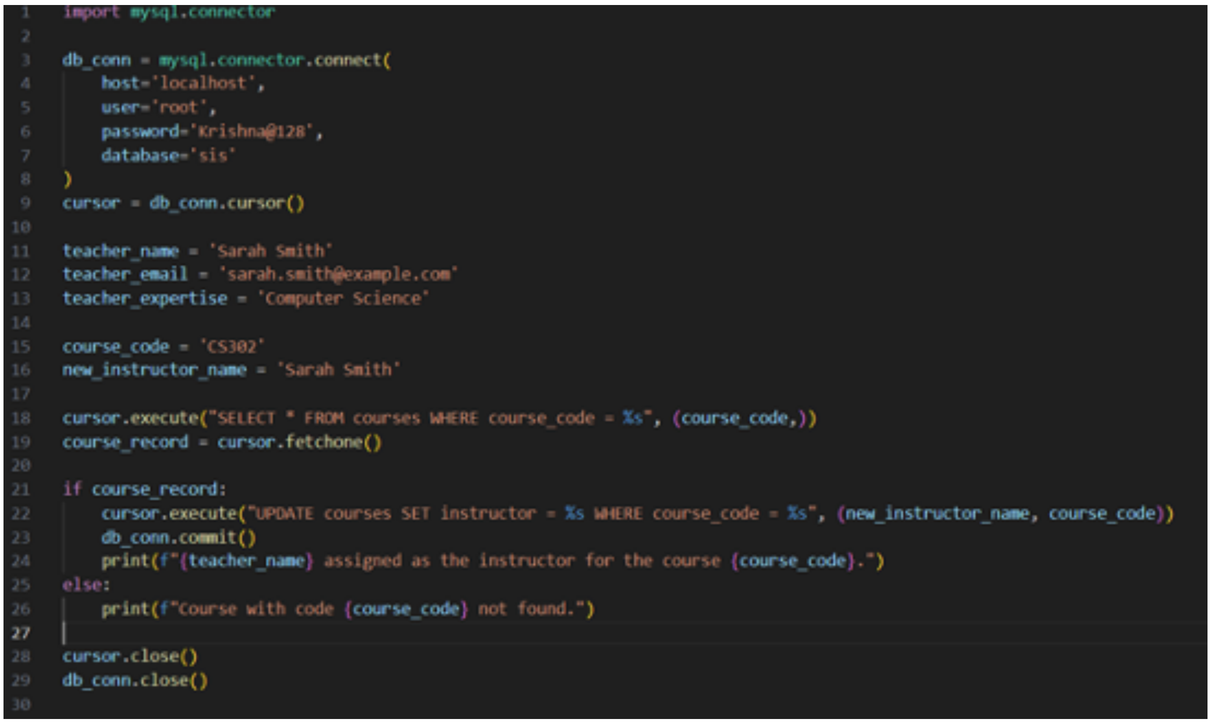
- Name: Sarah Smith
- Email: sarah.smith@example.com
- Expertise: Computer Science

Course to be assigned:

- Course Name: Advanced Database Management
- Course Code: CS302

The system should perform the following tasks:

- Retrieve the course record from the database based on the course code.
- Assign Sarah Smith as the instructor for the course.
- Update the course record in the database with the new instructor information.



```python
import mysql.connector

db_conn = mysql.connector.connect(
    host='localhost',
    user='root',
    password='Krishna@128',
    database='sis'
)
cursor = db_conn.cursor()

teacher_name = 'Sarah Smith'
teacher_email = 'sarah.smith@example.com'
teacher_expertise = 'Computer Science'

course_code = 'CS302'
new_instructor_name = 'Sarah Smith'

cursor.execute("SELECT * FROM courses WHERE course_code = %s", (course_code,))
course_record = cursor.fetchone()

if course_record:
    cursor.execute("UPDATE courses SET instructor = %s WHERE course_code = %s", (new_instructor_name, course_code))
    db_conn.commit()
    print(f"{teacher_name} assigned as the instructor for the course {course_code}.")
else:
    print(f"Course with code {course_code} not found.")

cursor.close()
db_conn.close()
```

Task 10: Payment Record

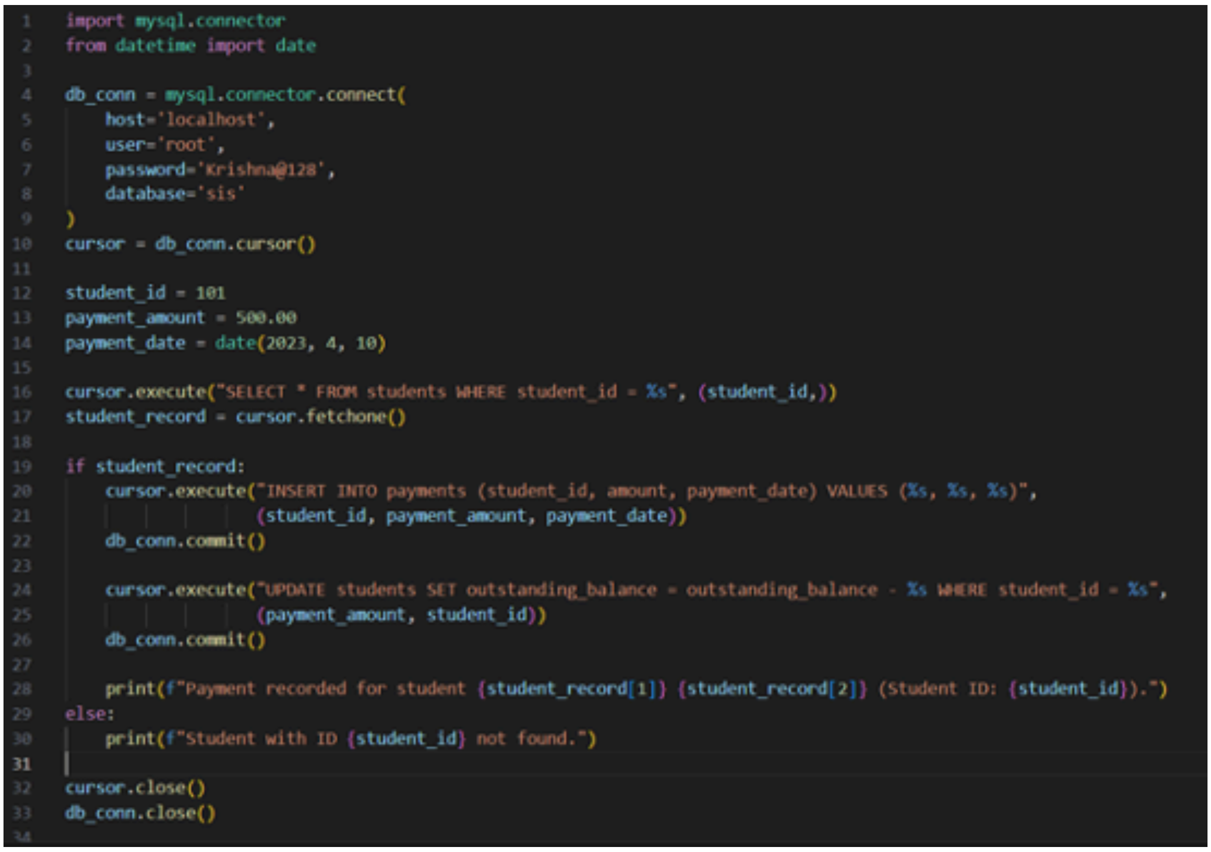
In this task, a student, Jane Johnson, makes a payment for her enrolled courses. The system needs to record this payment in the database.

Jane Johnson's details:

- Student ID: 101
- Payment Amount: $500.00
- Payment Date: 2023-04-10

The system should perform the following tasks:

- Retrieve Jane Johnson's student record from the database based on her student ID.
- Record the payment information in the database, associating it with Jane's student record.
- Update Jane's outstanding balance in the database based on the payment amount.



```python
import mysql.connector
from datetime import date

db_conn = mysql.connector.connect(
    host='localhost',
    user='root',
    password='Krishna@128',
    database='sis'
)
cursor = db_conn.cursor()

student_id = 101
payment_amount = 500.00
payment_date = date(2023, 4, 10)

cursor.execute("SELECT * FROM students WHERE student_id = %s", (student_id,))
student_record = cursor.fetchone()

if student_record:
    cursor.execute("INSERT INTO payments (student_id, amount, payment_date) VALUES (%s, %s, %s)",
                   (student_id, payment_amount, payment_date))
    db_conn.commit()

    cursor.execute("UPDATE students SET outstanding_balance = outstanding_balance - %s WHERE student_id = %s",
                   (payment_amount, student_id))
    db_conn.commit()

    print(f"Payment recorded for student {student_record[1]} {student_record[2]} (Student ID: {student_id}).")
else:
    print(f"Student with ID {student_id} not found.")

cursor.close()
db_conn.close()
```

Task 11: Enrollment Report Generation

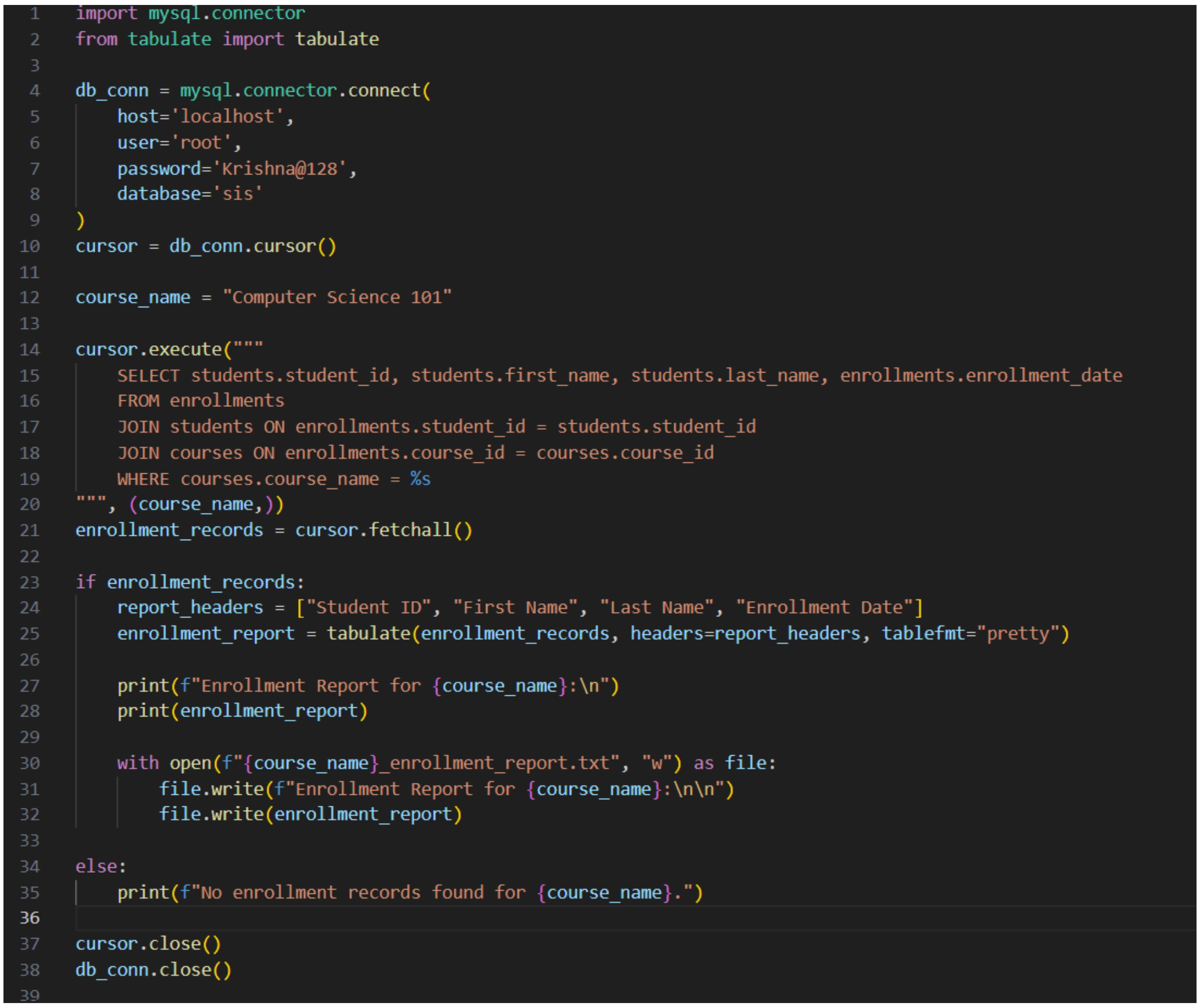
In this task, an administrator requests an enrollment report for a specific course, "Computer Science 101." The system needs to retrieve enrollment information from the database and generate a report.

Course to generate the report for:

- Course Name: Computer Science 101

The system should perform the following tasks:

- Retrieve enrollment records from the database for the specified course.
- Generate an enrollment report listing all students enrolled in Computer Science 101.
- Display or save the report for the administrator.



```python
import mysql.connector
from tabulate import tabulate

db_conn = mysql.connector.connect(
    host='localhost',
    user='root',
    password='Krishna@128',
    database='sis'
)
cursor = db_conn.cursor()

course_name = "Computer Science 101"

cursor.execute("""
    SELECT students.student_id, students.first_name, students.last_name, enrollments.enrollment_date
    FROM enrollments
    JOIN students ON enrollments.student_id = students.student_id
    JOIN courses ON enrollments.course_id = courses.course_id
    WHERE courses.course_name = %s
""", (course_name,))
enrollment_records = cursor.fetchall()

if enrollment_records:
    report_headers = ["Student ID", "First Name", "Last Name", "Enrollment Date"]
    enrollment_report = tabulate(enrollment_records, headers=report_headers, tablefmt="pretty")

    print(f"Enrollment Report for {course_name}:\n")
    print(enrollment_report)

    with open(f"{course_name}_enrollment_report.txt", "w") as file:
        file.write(f"Enrollment Report for {course_name}:\n\n")
        file.write(enrollment_report)

else:
    print(f"No enrollment records found for {course_name}.")

cursor.close()
db_conn.close()

```