Assignment -2 Python
Student Information System (SIS)

By- Kunal Sudhir Mishra

Implement OOPs :

A Student Information System (SIS) manages information about students, courses, student enrollments, teachers, andpayments. Each student can enroll in multiple courses, each course can have multiple students, each course is taughtby a teacher, and students make payments for their courses. Students have attributes such as name, date of birth, email, and phone number. Courses have attributes such as course name, course code, and instructor name. Enrollmentstrack which students are enrolled in which courses. Teachers have attributes such as names and email. Payments trackthe amount and date of payments made by students.

Task 1: Define Classes :

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

NumberCourse class with the following attributes:

• Course ID • Course Name • Course Code • Instructor Name

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

DateTeacher class with the following attributes:

• Teacher ID • First Name • Last Name •

EmailPayment class with the following

attributes:

• Payment ID • Student ID (reference to a Student) • Amount • Payment Date

Source Code:

```
4 usages                                                                    ⚠ 11
class Student:
    def __init__(self, Student_ID=None, First_Name=None, Last_Name=None, DOB=None, Email=None, Phone_NO=None):
        self.Student_ID = Student_ID
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.DOB = DOB
        self.Email = Email
        self.Phone_NO = Phone_NO

2 usages
class Course:
    def __init__(self, Course_ID=None, Course_Name=None, Course_Code=None, Instructor_Name=None):
        self.Course_ID = Course_ID
        self.Course_Name = Course_Name
        self.Course_Code = Course_Code
        self.Instructor_Name = Instructor_Name
```

```python
class Enrollment(Student, Course):
    def __init__(self, Enrollment_ID=None, Student_ID=None, Course_ID=None, Enrollment_Date=None):
        Student.__init__(self, Student_ID)
        Course.__init__(self, Course_ID)
        self.Enrollment_ID = Enrollment_ID
        self.Enrollment_Date = Enrollment_Date

class Teacher:
    def __init__(self, Teacher_ID=None, First_Name=None, Last_Name=None, Email=None):
        self.Teacher_ID = Teacher_ID
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.Email = Email

class Payment(Student):
    def __init__(self, Payment_ID=None, Student_ID=None, Amount=None, Payment_Date=None):
        self.Payment_ID = Payment_ID
        Student.__init__(self, Student_ID)
        self.Amount = Amount
        self.Payment_Date = Payment_Date
```

Task 2: Implement Constructors

Implement constructors for each class to initialize their attributes. Constructors are special methods that are called when an object of a class is created. They are used to set initial values for the attributes of the class. Below are detailed instructions on how to implement constructors for each class in your Student

Information System (SIS) assignment:

Student Class Constructor

In the Student class, you need to create a constructor that initializes the attributes of a student when an instance ofthe Student class is created

SIS Class Constructor

If you have a class that represents the Student Information System itself (e.g., SIS class), you may also implement a constructor for it. This constructor can be used to set up any initial configuration for the SIS.

Repeat the above process for each class Course, Enrollment, Teacher, Payment by defining constructors that initialize their respective attributes.

Source Code:

```
class Student:
    def __init__(self, Student_ID=None, First_Name=None, Last_Name=None, DOB=None, Email=None, Phone_NO=None):
        self.Student_ID = Student_ID
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.DOB = DOB
        self.Email = Email
        self.Phone_NO = Phone_NO

3 usages
class Course:
    def __init__(self, Course_ID=None, Course_Name=None, Course_Code=None, Instructor_Name=None):
        self.Course_ID = Course_ID
        self.Course_Name = Course_Name
        self.Course_Code = Course_Code
        self.Instructor_Name = Instructor_Name

class Enrollment(Student, Course):
    def __init__(self, Enrollment_ID=None, Student_ID=None, Course_ID=None, Enrollment_Date=None):
        Student.__init__(self, Student_ID=Student_ID)
        Course.__init__(self, Course_ID=Course_ID)
        self.Enrollment_ID = Enrollment_ID
        self.Enrollment_Date = Enrollment_Date
```

```python
class Teacher:
    def __init__(self, Teacher_ID=None, First_Name=None, Last_Name=None, Email=None):
        self.Teacher_ID = Teacher_ID
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.Email = Email

class Payment(Student):
    def __init__(self, Payment_ID=None, Student_ID=None, Amount=None, Payment_Date=None):
        self.Payment_ID = Payment_ID
        Student.__init__(self, Student_ID)
        self.Amount = Amount
        self.Payment_Date = Payment_Date

class SIS:
    def __init__(self):
        self.students = []
        self.teachers = []
        self.courses = []
        self.payments = []
```

Task 3: Implement Methods

Implement methods in your classes to perform various operations related to the Student Information System (SIS). These methods will allow you to interact with and manipulate data within your system. Below are detailed instructions on how to implement methods in each class:

Implement the following methods in the appropriate classes:

Student Class:

- **EnrollInCourse(course: Course): Enrolls the student in a course.**
- **UpdateStudentInfo(firstName: string, lastName: string, dateOfBirth: DateTime, email: string,phoneNumber: string): Updates the student's information.**
- **MakePayment(amount: decimal, paymentDate: DateTime): Records a payment made by thestudent.**
- **DisplayStudentInfo(): Displays detailed information about the student.**
- **GetEnrolledCourses(): Retrieves a list of courses in which the student is enrolled.**
- **GetPaymentHistory(): Retrieves a list of payment records for the student.**

Course Class:

- **AssignTeacher(teacher: Teacher): Assigns a teacher to the course.**
- **UpdateCourseInfo(courseCode: string, courseName: string, instructor: string): Updates courseinformation.**
- **DisplayCourseInfo(): Displays detailed information about the course.**
- **GetEnrollments(): Retrieves a list of student enrollments for the course.**
- **GetTeacher(): Retrieves the assigned teacher for the course.**

Enrollment Class:

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

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.**

Payment Class:

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

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 apayment made by a student.

• GenerateEnrollmentReport(course: Course): Generates a report of students enrolled in

aspecific course.

• GeneratePaymentReport(student: Student): Generates a report of payments made by a

specificstudent.

• CalculateCourseStatistics(course: Course): Calculates statistics for a specific course, such as

thenumber of enrollments and total payments.

Source Code:

```
class Student:
    def __init__(self, Student_ID=None, First_Name=None, Last_Name=None, DOB=None, Email=None, Phone_NO=None):
        self.Student_ID = Student_ID
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.DOB = DOB
        self.Email = Email
        self.Phone_NO = Phone_NO
        self.enrolled_courses = []
        self.payment_made = []

    def EnrollInCourse(self, course):
        if course not in self.enrolled_courses:
            self.enrolled_courses.append(course)
            print(f"Student {self.Student_ID} enrolled in course {course.Course_ID}")
        else:
            print(f"Student {self.Student_ID} is already enrolled in course {course.Course_ID}")

    def UpdateStudentInfo(self, First_Name, Last_Name, DOB, Email, Phone_NO):
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.DOB = DOB
        self.Email = Email
```

```python
    def MakePayment(self, payment):
        pass
4 usages
class Student:
    def __init__(self, Student_ID=None, First_Name=None, Last_Name=None, DOB=None, Email=None, Phone_NO=None):
        self.Student_ID = Student_ID
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.DOB = DOB
        self.Email = Email
        self.Phone_NO = Phone_NO
        self.enrolled_courses = []
        self.payment_made = []

    def EnrollInCourse(self, course):
        if course not in self.enrolled_courses:
            self.enrolled_courses.append(course)
            print(f"Student {self.Student_ID} enrolled in course {course.Course_ID}")
        else:
            print(f"Student {self.Student_ID} is already enrolled in

    def UpdateStudentInfo(self, First_Name, Last_Name, DOB, Email, Phone_NO):
        self.First_Name = First_Name
        self.Last_Name = Last_Name
```

Parameter **course** of main.Student.EnrollInCou
course: Any

```python
        self.DOB = DOB
        self.Email = Email
        self.Phone_NO = Phone_NO

    def MakePayment(self, payment):
        self.payment_made.append(payment)
        print(f"Payment of {payment.Amount} made by {self.First_Name} on {payment.Payment_Date}")

    def DisplayStudentInfo(self):
        print('-' * 50)
        print("Students Detail: ")
        print(f"Student ID: {self.Student_ID}")
        print(f"Name: {self.First_Name} {self.Last_Name}")
        print(f"Date of Birth: {self.DOB}")
        print(f"Email: {self.Email}")
        print(f"Phone Number: {self.Phone_NO}")

    def GetEnrolledCourses(self):
        print('-' * 50)
        print(f"Enrolled Courses for {self.Student_ID}:")
        for course in self.enrolled_courses:
            print(f"{course.Course_ID}-{course.Course_Name}")
```

```
    def GetPaymentHistory(self):
        print(f"Payment History for {self.Student_ID}: ")
        for payment in self.payment_made:
            print(f"Payment Amount: {payment.Amount}, Date: {payment.Payment_Date} ")


2 usages
class Course:
    def __init__(self, Course_ID=None, Course_Name=None, Course_Code=None, Instructor_Name=None):
        self.Course_ID = Course_ID
        self.Course_Name = Course_Name
        self.Course_Code = Course_Code
        self.Instructor_Name = Instructor_Name
        self.course_assigned = []
        self.enrolled_students = []

    def AssignTeacher(self, teacher):
        if teacher not in self.course_assigned:
            self.course_assigned.append(teacher)
            print(f"{teacher.First_Name} has been assigned to the course {self.Course_ID}")
        else:
            print(f"{teacher.First_Name} is already assigned to the course {self.Course_ID}")
```

```
class Student:
    def __init__(self, Student_ID=None, First_Name=None, Last_Name=None, DOB=None, Email=None, Phone_NO=None):
        self.Student_ID = Student_ID
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.DOB = DOB
        self.Email = Email
        self.Phone_NO = Phone_NO
        self.enrolled_courses = []
        self.payment_made = []

    1 usage (1 dynamic)
    def EnrollInCourse(self, course):
        if course not in self.enrolled_courses:
            self.enrolled_courses.append(course)
            print(f"Student {self.Student_ID} enrolled in course {course.Course_ID}")
        else:
            print(f"Student {self.Student_ID} is already enrolled in course {course.Course_ID}")

    def UpdateStudentInfo(self, First_Name, Last_Name, DOB, Email, Phone_NO):
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.DOB = DOB
```

```
        self.Email = Email
        self.Phone_NO = Phone_NO

    1 usage (1 dynamic)
    def MakePayment(self, payment):
        self.payment_made.append(payment)
        print(f"Payment of {payment.Amount} made by {self.First_Name} on {payment.Payment_Date}")

    def DisplayStudentInfo(self):
        print('-' * 50)
        print("Students Detail: ")
        print(f"Student ID: {self.Student_ID}")
        print(f"Name: {self.First_Name} {self.Last_Name}")
        print(f"Date of Birth: {self.DOB}")
        print(f"Email: {self.Email}")
        print(f"Phone Number: {self.Phone_NO}")

    def GetEnrolledCourses(self):
        print('-' * 50)
        print(f"Enrolled Courses for {self.Student_ID}:")
        for course in self.enrolled_courses:
            print(f"{course.Course_ID}-{course.Course_Name}")
```

```
    def UpdateCourseInfo(self, Course_Name, Course_Code, Instructor_Name):
        self.Course_Name = Course_Name
        self.Course_Code = Course_Code
        self.Instructor_Name = Instructor_Name

    def DisplayCourseInfo(self):
        print('-' * 50)
        print("Course Detail: ")
        print("Course Id: ", self.Course_ID)
        print("Course Name: ", self.Course_Name)
        print("Course Code: ", self.Course_Code)
        print("Instructor Name: ", self.Instructor_Name)

    def GetEnrollments(self, student):
        print("Enrolled Students")
        for student in self.enrolled_students:
            print(f"Student {student.First_Name} has enrolled in {self.Course_Code} course")

    def GetTeacher(self, teacher):
        print("Assigned Teacher")
        for teacher in self.course_assigned:
            print(f"Teacher {teacher.First_Name} has assigned to {self.Course_Code} course")
```

```python
class Enrollment(Student, Course):
    def __init__(self, Enrollment_ID=None, Student_ID=None, Course_ID=None, Enrollment_Date=None):
        Student.__init__(self, Student_ID)
        Course.__init__(self, Course_ID)
        self.Enrollment_ID = Enrollment_ID
        self.Enrollment_Date = Enrollment_Date

    def GetStudent(self):
        return self.Student_ID

    def GetCourse(self):
        return self.Course_ID


class Teacher:
    def __init__(self, Teacher_ID=None, First_Name=None, Last_Name=None, Email=None):
        self.Teacher_ID = Teacher_ID
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.Email = Email
        self.assigned_courses = []
```

```python
        print('-' * 50)
        print("Teacher Info: ")
        print(f"Teacher ID: {self.Teacher_ID}")
        print(f"Name: {self.First_Name} {self.Last_Name}")
        print(f"Email: {self.Email}")

    def UpdateTeacherInfo(self, First_Name, Last_Name, Email):
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.Email = Email

    def AssignToCourse(self, course):
        if course not in self.assigned_courses:
            self.assigned_courses.append(course)
            print('-' * 50)
            print(f"Teacher {self.First_Name} assigned to course {course.Course_Name}")
        else:
            print('-' * 50)
            print(f"Teacher {self.First_Name} is already assigned to course {course.Course_Name}")

    def GetAssignedCourses(self):
        print('-' * 50)
        print(f"Courses assigned to Teacher {self.First_Name}:")
        for course in self.assigned_courses:
```

```
    def GetPaymentHistory(self):
        print(f"Payment History for {self.Student_ID}: ")
        for payment in self.payment_made:
            print(f"Payment Amount: {payment.Amount}, Date: {payment.Payment_Date} ")


4 usages
class Course:
    def __init__(self, Course_ID=None, Course_Name=None, Course_Code=None, Instructor_Name=None):
        self.Course_ID = Course_ID
        self.Course_Name = Course_Name
        self.Course_Code = Course_Code
        self.Instructor_Name = Instructor_Name
        self.course_assigned = []
        self.enrolled_students = []

    1 usage (1 dynamic)
    def AssignTeacher(self, teacher):
        if teacher not in self.course_assigned:
            self.course_assigned.append(teacher)
            print(f"{teacher.First_Name} has been assigned to the course {self.Course_ID}")
        else:
            print(f"{teacher.First_Name} is already assigned to the course {self.Course_ID}")
```

```
            print(f"{course.Course_ID}-{course.Course_Name}")


 1 usage
 class Payment(Student):
     def __init__(self, Payment_ID=None, Student_ID=None, Amount=None, Payment_Date=None):
         self.Payment_ID = Payment_ID
         Student.__init__(self, Student_ID)
         self.Amount = Amount
         self.Payment_Date = Payment_Date

     def GetStudent(self):
         print(f"{self.Student_ID} has made the payment")

     def GetPaymentAmount(self):
         print(f"Payment of {self.Amount} has been made")

     def GetPaymentDate(self):
         print(f"The Payment was made on {self.Payment_Date} ")
```

```python
class SIS:
    def __init__(self):
        self.students = []
        self.teachers = []
        self.courses = []
        self.payments = []

    def EnrollStudentInCourse(self, student, course):
        student.EnrollInCourse(course)

    def AssignTeacherToCourse(self, teacher, course):
        course.AssignTeacher(teacher)

    def RecordPayment(self, student, amount, payment_date):
        payment = Payment(Student_ID=student.Student_ID, Amount=amount, Payment_Date=payment_date)
        student.MakePayment(payment)
        self.payments.append(payment)

    def GenerateEnrollmentReport(self, course):
        print(f"Enrollment Report for Course {course.Course_Name}:")
        for student in course.enrolled_students:
            print(f"Student ID: {student.Student_ID}, Name: {student.First_Name} {student.Last_Name}")
```

```python
    def GenerateEnrollmentReport(self, course):
        print(f"Enrollment Report for Course {course.Course_Name}:")
        for student in course.enrolled_students:
            print(f"Student ID: {student.Student_ID}, Name: {student.First_Name} {student.Last_Name}")

    def GeneratePaymentReport(self, student):
        print(f"Payment Report for Student {student.Student_ID} - {student.First_Name} {student.Last_Name}:")
        for payment in student.payment_made:
            print(f"Payment Amount: {payment.Amount}, Date: {payment.Payment_Date}")

    def CalculateCourseStatistics(self, course):
        enrolled_students_count = len(course.enrolled_students)
        total_payments = sum(payment.Amount for student in course.enrolled_students for payment in student.payment_made)
        print('-' * 50)
        print(f"Statistics for Course {course.Course_Name}:")
        print(f"Number of Enrollments: {enrolled_students_count}")
        print(f"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 businesslogic 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 derivedclasses (e.g., System.ApplicationException). These custom exception classes will allow you to encapsulate specific errorscenarios 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. Thisexception can be used in the EnrollStudentInCourse method.

• CourseNotFoundException: Thrown when a course does not exist in the system, and you attempt to performoperations 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 performoperations 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 toa course.

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

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

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

• InvalidEnrollmentDataException: Thrown when data provided for creating an enrollment is invalid (e.g., missingstudent or course references).

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

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

Source Code:

```python
class DuplicateEnrollmentException(Exception):
    def __init__(self, student_id, course_id):
        super().__init__(f"Student {student_id} is already enrolled in course {course_id}")


class CourseNotFoundException(Exception):
    def __init__(self, course_id):
        super().__init__(f"Course {course_id} not found")


class StudentNotFoundException(Exception):
    def __init__(self, student_id):
        super().__init__(f"Student {student_id} not found")


class TeacherNotFoundException(Exception):
    def __init__(self, teacher_id):
        super().__init__(f"Teacher {teacher_id} not found")


class PaymentValidationException(Exception):
    def __init__(self, message):
        super().__init__(f"Payment validation failed: {message}")
```

```python
class InvalidStudentDataException(Exception):
    def __init__(self, message):
        super().__init__(f"Invalid student data: {message}")


class InvalidCourseDataException(Exception):
    def __init__(self, message):
        super().__init__(f"Invalid course data: {message}")


class InvalidEnrollmentDataException(Exception):
    def __init__(self, message):
        super().__init__(f"Invalid enrollment data: {message}")


class InvalidTeacherDataException(Exception):
    def __init__(self, message):
        super().__init__(f"Invalid teacher data: {message}")


class InsufficientFundsException(Exception):
    def __init__(self, student_id, course_id):
        super().__init__(f"Student {student_id} does not have enough funds to enroll in course {course_id}")
```

```python
class Student:                                                    # 18
    def __init__(self, Student_ID=None, First_Name=None, Last_Name=None, DOB=None, Email=None, Phone_NO=None):
        self.Student_ID = Student_ID
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.DOB = DOB
        self.Email = Email
        self.Phone_NO = Phone_NO
        self.enrolled_courses = []
        self.payment_made = []

    1 usage (1 dynamic)
    def EnrollInCourse(self, course):
        if course not in self.enrolled_courses:
            self.enrolled_courses.append(course)
            print(f"Student {self.Student_ID} enrolled in course {course.Course_ID}")
        else:
            print(f"Student {self.Student_ID} is already enrolled in course {course.Course_ID}")

    def UpdateStudentInfo(self, First_Name, Last_Name, DOB, Email, Phone_NO):
        self.First_Name = First_Name
        self.Last_Name = Last_Name
        self.DOB = DOB
        self.Email = Email
        self.Phone_NO = Phone_NO

1 usage (1 dynamic)
def MakePayment(self, payment):
    self.payment_made.append(payment)
    print(f"Payment of {payment.Amount} made by {self.First_Name} on {payment.Payment_Date}")

def DisplayStudentInfo(self):
    print('-' * 50)
    print("Students Detail: ")
    print(f"Student ID: {self.Student_ID}")
    print(f"Name: {self.First_Name} {self.Last_Name}")
    print(f"Date of Birth: {self.DOB}")
    print(f"Email: {self.Email}")
    print(f"Phone Number: {self.Phone_NO}")

def GetEnrolledCourses(self):
    print('-' * 50)
    print(f"Enrolled Courses for {self.Student_ID}:")
    for course in self.enrolled_courses:
        print(f"{course.Course_ID}-{course.Course_Name}")

def GetPaymentHistory(self):
    print(f"Payment History for {self.Student_ID}: ")
```

```python
class Course:
    def __init__(self, Course_ID=None, Course_Name=None, Course_Code=None, Instructor_Name=None):
        self.Course_ID = Course_ID
        self.Course_Name = Course_Name
        self.Course_Code = Course_Code
        self.Instructor_Name = Instructor_Name
        self.course_assigned = []
        self.enrolled_students = []

    1 usage (1 dynamic)
    def AssignTeacher(self, teacher):
        if teacher not in self.course_assigned:
            self.course_assigned.append(teacher)
            print(f"{teacher.First_Name} has been assigned to the course {self.Course_Name}")
        else:
            print(f"{teacher.First_Name} has already been assigned to this course")

    def AssignStudent(self, student):
        if student not in self.course_assigned:
            self.enrolled_students.append(student)
            print(f"{student.First_Name} has been enrolled to the course {self.Course_Name}")
        else:
```

```python
    def UpdateCourseInfo(self, Course_Name, Course_Code, Instructor_Name):
        self.Course_Name = Course_Name
        self.Course_Code = Course_Code
        self.Instructor_Name = Instructor_Name

    def DisplayCourseInfo(self):
        print('-' * 50)
        print("Course Detail: ")
        print("Course Id: ", self.Course_ID)
        print("Course Name: ", self.Course_Name)
        print("Course Code: ", self.Course_Code)
        print("Instructor Name: ", self.Instructor_Name)

    def GetEnrollments(self, student):
        print("Enrolled Students")
        for student in self.enrolled_students:
            print(f"Student {student.First_Name} has enrolled in {self.Course_Code} course")

    def GetTeacher(self, teacher):
        print("Assigned Teacher")
        for teacher in self.course_assigned:
            print(f"Teacher {teacher.First_Name} has assigned to {self.Course_Code} course")
```

```python
class Enrollment(Student, Course):
    def __init__(self, Enrollment_ID=None, Student_ID=None, Course_ID=None, Enrollment_Date=None):
        Student.__init__(self, Student_ID)
        Course.__init__(self, Course_ID)
        self.Enrollment_ID = Enrollment_ID
        self.Enrollment_Date = Enrollment_Date

    def GetStudent(self):
        return self.Student_ID

    def GetCourse(self):
        return self.Course_ID
```

Task 5: Collections

Implement Collections:

Implement relationships between classes using appropriate data structures (e.g., lists or dictionaries) to maintainassociations between students, courses, enrollments, teachers, and payments.

These relationships are essential for the Student Information System (SIS) to track and manage student enrollments,teacher assignments, and payments accurately.

Define Class-Level Data Structures

You will need class-level data structures within each class to maintain relationships. Here's how to define them for eachclass:

Update Constructor(s)

In the constructors of your classes, initialize the list or collection properties to create empty collections

when an object is instantiated.

Repeat this for the Course, Teacher, and Payment classes, where applicable.

Student Class:

Create a list or collection property to store the student's enrollments. This property will hold references to Enrollmentobjects.

Example: List<Enrollment> Enrollments { get; set; }

Course Class:

Create a list or collection property to store the course's enrollments. This property will hold

referencesto Enrollment objects.

Example: List<Enrollment> Enrollments { get; set; }

Enrollment Class:

Include properties to hold references to both the Student and Course objects.

Example: Student Student { get; set; } and Course Course { get; set; }

Teacher Class:

Create a list or collection property to store the teacher's assigned courses. This property will hold references toCourse objects.

Example: List<Course> AssignedCourses { get; set; }

Payment Class:

Include a property to hold a reference to the Student object.

Example: Student Student { get; set; }

```python
from Student import Student
from Course import Course
from Teacher import Teacher
from Payment import Payment
from Exceptions import *

class SIS:
    def __init__(self):
        self.students = []
        self.teachers = []
        self.courses = []
        self.payments = []

    def EnrollStudentInCourse(self, student, course):
        self.students.append(student)
        self.courses.append(course)
        course.enrolled_students.append(student)

        if course not in self.courses:
            raise CourseNotFoundException(f"{course.Course_ID} not found")
        if not isinstance(student, Student):
            raise InvalidStudentDataException("Invalid student object provided")
        if student in self.students:
            raise DuplicateEnrollmentException(student.Student_ID, course.Course_ID)
```

```
        if not isinstance(course, Course):
            raise InvalidCourseDataException("Invalid course object provided")

        try:
            student.EnrollInCourse(course)
            self.students.append(student)
        except DuplicateEnrollmentException as e:
            print(f"Enrollment Error: {e}")
            raise e
        except Exception as e:
            print(f"Unexpected Error: {e}")
            raise e

    def AssignTeacherToCourse(self, teacher, course):
        self.courses.append(course)

        if teacher not in self.teachers:
            raise TeacherNotFoundException(teacher.Teacher_ID)
        if type(teacher) is not Teacher:
            raise InvalidTeacherDataException("Invalid teacher object provided")

        if type(course) is not Course:
            raise InvalidCourseDataException("Invalid course object provided")
```

```
        try:
            course.AssignTeacher(teacher)
        except TeacherNotFoundException as e:
            raise e  # Re-raise the specific exception
        except Exception as e:
            raise e  # Re-raise any unexpected exceptions

    def RecordPayment(self, student, amount, payment_date):
        if student not in self.students:
            raise StudentNotFoundException(student.Student_ID)
        if type(student) is not Student:
            raise InvalidStudentDataException("Invalid student object provided")
        if amount < 0:
            raise PaymentValidationException("Invalid Amount")

        try:
            payment = Payment(Student_ID=student.Student_ID, Amount=amount, Payment_Date=payment_date)
            student.MakePayment(payment)
            self.payments.append(payment)
        except PaymentValidationException as e:
            raise e
        except Exception as e:
            raise e
```

```python
def GenerateEnrollmentReport(self, course):
    print(f"Enrollment Report for Course {course.Course_Name}:")
    for student in course.enrolled_students:
        print(f"Student ID: {student.Student_ID}, Name: {student.First_Name} {student.Last_Name}")

def CalculateCourseStatistics(self, course):
    enrolled_students_count = len(course.enrolled_students)
    total_payments = sum(payment.Amount for student in course.enrolled_students for payment in student.payment_made)
    print('-' * 50)
    print(f"Statistics for Course {course.Course_Name}:")
    print(f"Number of Enrollments: {enrolled_students_count}")
    print(f"Total Payments: {total_payments}")

def AddEnrollment(self, student, course):
    self.students.append(student)
    student.enrollments.append(student)
    course.enrollments.append(course)

def AssignCourseToTeacher(self, course, teacher):
    self.teachers.append(teacher)
    course.AssignTeacher(teacher)
```

```python
def AssignCourseToTeacher(self, course, teacher):
    self.teachers.append(teacher)
    course.AssignTeacher(teacher)

def AddPayment(self, payment, student):
    self.payments.append(payment)
    student.MakePayment(payment)

def GetEnrollmentsForStudent(self, student):
    if student not in self.students:
        raise StudentNotFoundException
```

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 boththe 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 thecourse to the teacher's AssignedCourses list.

• AddPayment(student, amount, paymentDate): In the SIS class, create a method that adds a payment to the Student'spayment 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.Source Code:

```
class SIS:
    def __init__(self):
        self.students = []
        self.teachers = []
        self.courses = []
        self.payments = []

    def EnrollStudentInCourse(self, student, course):
        self.students.append(student)
        self.courses.append(course)
        course.enrolled_students.append(student)

        if course not in self.courses:
            raise CourseNotFoundException(f"{course.Course_ID} not found")
        if not isinstance(student, Student):
            raise InvalidStudentDataException("Invalid student object provided")
        if student in self.students:
            raise DuplicateEnrollmentException(student.Student_ID, course.Course_ID)
        if not isinstance(course, Course):
            raise InvalidCourseDataException("Invalid course object provided")

        try:
```

```
    def AssignTeacherToCourse(self, teacher, course):
        self.courses.append(course)

        if teacher not in self.teachers:
            raise TeacherNotFoundException(teacher.Teacher_ID)
        if type(teacher) is not Teacher:
            raise InvalidTeacherDataException("Invalid teacher object provided")

        if type(course) is not Course:
            raise InvalidCourseDataException("Invalid course object provided")

        try:
            course.AssignTeacher(teacher)
        except TeacherNotFoundException as e:
            raise e  # Re-raise the specific exception
        except Exception as e:
            raise e  # Re-raise any unexpected exceptions

    def RecordPayment(self, student, amount, payment_date):
        if student not in self.students:
            raise StudentNotFoundException(student.Student_ID)
        if type(student) is not Student:
            raise InvalidStudentDataException("Invalid student object provided")
        if amount < 0:
            raise PaymentValidationException("Invalid Amount")
```

```
        try:
            payment = Payment(Student_ID=student.Student_ID, Amount=amount, Payment_Date=payment_date)
            student.MakePayment(payment)
            self.payments.append(payment)
        except PaymentValidationException as e:
            raise e
        except Exception as e:
            raise e

    def GeneratePaymentReport(self, student):
        print(f"Payment Report for Student {student.Student_ID} - {student.First_Name} {student.Last_Name}:")
        for payment in student.payment_made:
            print(f"Payment Amount: {payment.Amount}, Date: {payment.Payment_Date}")

    def GenerateEnrollmentReport(self, course):
        print(f"Enrollment Report for Course {course.Course_Name}:")
        for student in course.enrolled_students:
            print(f"Student ID: {student.Student_ID}, Name: {student.First_Name} {student.Last_Name}")

    def CalculateCourseStatistics(self, course):
        enrolled_students_count = len(course.enrolled_students)
        total_payments = sum(
            payment.Amount for student in course.enrolled_students for payment in student.payment_made)
```

```
        print(f"Number of Enrollments: {enrolled_students_count}")
        print(f"Total Payments: {total_payments}")

    def AddEnrollment(self, student, course):
        self.students.append(student)
        student.enrollments.append(student)
        course.enrollments.append(course)

    def AssignCourseToTeacher(self, course, teacher):
        self.teachers.append(teacher)
        course.AssignTeacher(teacher)

    def AddPayment(self, payment, student):
        self.payments.append(payment)
        student.MakePayment(payment)

    def GetEnrollmentsForStudent(self, student):
        if student not in self.students:
            raise StudentNotFoundException(student.Student_ID)
        print(f"Enrollments for Student {student.First_Name} {student.Last_Name}:")

        enrolled_courses = [course for course in student.enrolled_courses]
        for course in enrolled_courses:
            print(f"{course.Course_ID}-{course.Course_Name}-{course.Course_Code}")
```

```
    def GetEnrollmentsForStudent(self, student):
        if student not in self.students:
            raise StudentNotFoundException(student.Student_ID)
        print(f"Enrollments for Student {student.First_Name} {student.Last_Name}:")

        enrolled_courses = [course for course in student.enrolled_courses]
        for course in enrolled_courses:
            print(f"{course.Course_ID}-{course.Course_Name}-{course.Course_Code}")

    def GetCoursesForTeacher(self, teacher):
        if teacher not in self.teachers:
            raise TeacherNotFoundException(teacher.Teacher_ID)

        teacher_courses = [course for course in teacher.assigned_courses]
        print('-' * 50)
        print(f"Courses assigned to Teacher {teacher.First_Name} {teacher.Last_Name}:")
        for course in teacher_courses:
            print(f"{course.Course_ID}-{course.Course_Name}")
```

```python
try:
    conn = mysql.connector.connect(host='localhost', user='root', passwd='root', database='sisdb', port='3306')
    if conn:
        print(f"Connected Successfully to {conn.database}")
except Exception as e:
    print(e)

cur = conn.cursor()
cur.execute("show tables")
tables = cur.fetchall()
print(f"Listing tables: ")
for i in tables:
    print(i)

# Data Retrieval
cur1 = conn.cursor()
cur1.execute("select * from courses")
print(f"Listing courses: ")
courses = cur1.fetchall()
for j in courses:
    print(j)
```

```python
# Data Retrieval
cur1 = conn.cursor()
cur1.execute("select * from courses")
print(f"Listing courses: ")
courses = cur1.fetchall()
for j in courses:
    print(j)

cur2 = conn.cursor()
cur2.execute("select first_name,course_id from students s join enrollments e on e.student_id=s.student_id")
enrollments = cur2.fetchall()
print("Listing The students who have been enrolled: ")
for stud in enrollments:
    print(stud)

print("\nTrying to execute query with the wrong table name:")
try:
```

```python
print("\nTrying to execute query with the wrong table name:")
try:
    cur3 = conn.cursor()
    cur3.execute("select s.first_name,p.amount,p.payment_date from payment p "
                 "join students s on p.student_id=s.student_id")
    payment_details = cur3.fetchall()
    print("Payment details for students: ")
    for pay in payment_details:
        print(pay)
except Exception as e:
    print(e)
```