**OBJECTIVE:** Starting with your design from phase one of this project, develop a Python application that will allow you to perform basic operations on the documents in the collections that you have designed, while ensuring that the business rules that apply are properly enforced.

**INTRODUCTION:** Throughout this course we have studied design patterns that properly capture and implement a variety of types of business rules. Some of these business rules can be implemented directly by the database management system, such as "every department must have a chairperson identified". Others, such as "we cannot have two sections schedules **overlapping** in the same building and room during the same semester and year" is complex enough that third generation language code is required to implement it. Remember, our uniqueness constraint only protects against multiple sections **starting** at the same time on the same day of the week, during the same semester and year in the same building and room.

Just lately, we have looked at tools in MongoDB that we can use for implementing business rules. Because the document-based nature of MongoDB favors embedding (combining) data together rather than normalizing (resolving subkeys) the way that we do in the relational paradigm, the ways in which we can enforce business rules in MongoDB differ from the ways that we would to so in a relational environment, such as PostgreSQL.

This assignment asks you to create a console application in Python that implements the design that you documented in phase 1 of this term project. Some of the business rules will be implemented by the schema that you impose on your collections, some will be implemented by unique indexes in MongoDB, and others will have to be implemented by your Python code. In general, **please use MongoDB** to implement as many of the business rules that your selected database design will allow you.

If you find that your initial design makes it difficult to implement many of the business rules (see below), you are free to alter your design. If you do so, I will require that you submit a revised copy of your design document with phase two of the term project.

Please note that you will be using the **same teams** that you used for **phase one** of this project. If you find that you are unable to continue working in the team that you were in for phase one, please let me know and we will work out a solution.

* You will use MongoEngine as your Object Document Mapping (ODM) layer on top of the PyMongo drivers for MongoDB.
  + Instead of using a validator (schema) to implement our field constraints the way that we did in the *MongoDB Schema and Uniqueness Constraints* project, we will use the declaration of our Python Document classes that MongoEngine gives us.
  + The MongoEngine meta specification in each class spells out the uniqueness constraints, which are implemented by MongoEngine as unique indexes in MongoDB.
* Please continue to use unique\_general from the sample code for the *MongoDB One to Many* assignment to prevent the user from creating redundant documents.
* Please use the Python try/except block for all other errors. This “hybrid” approach will give you the most control over the feedback to the user when there is an error in the input data.

**PROCEDURE:** I am going to limit the business rules that you are responsible for to a subset of all the business rules that you implemented in SQLAlchemy. This will make the overall assignment less time-consuming, and it will make your demonstration video shorter. A summary of the business rules that I am going to look for when I grade your work follows:

1. In general –
   1. All the attributes shown in the SQLAlchemy Project Roadmap model are required. You will find that roadmap model in the Canvas Term Project module. A quick shortcut to get to that module is found on the home page:

A screenshot of a computer

Description automatically generated with medium confidence

* 1. All the attributes must have the datatype as shown in the SQLAlchemy Project Roadmap model.

1. Associations
   1. In general
      1. As you know, MongoDB does not have referential integrity constraints in the same sense that a relational database management system does.
         1. We will **not** update any document attributes, so we do not have to worry about maintaining referential integrity when the parent primary key is updated.
         2. If one collection references another collection, you will need to make sure that the "child" collection references a valid document in the "parent" collection when the child is inserted.
      2. We **will**, however, make sure that the child is never left an orphan.
         1. If you decide to embed one class in another in your design, that makes referential integrity simpler. Deleting a document will remove both the parent and all the children without any action needed on your part.
         2. On the other hand, you need to protect the child documents from getting orphaned by the deletion of their parent document. MongoEngine offers the reverse\_delete\_rule attribute in reference attributes. Reference the user guide [here](https://docs.mongoengine.org/guide/defining-documents.html#reference-fields) for how to use that attribute.
   2. Each Course belongs to exactly one Department.
   3. Each Major belongs to exactly one Department.
   4. Each Student can declare many Majors and each Major can be declared by many Students.
   5. Each Section belongs to exactly one Course.
   6. Each Student can enroll in many sections and each Section can be enrolled in by many students.
   7. Each Enrolment must either fall into the PassFail category, or the LetterGrade category.
      1. Please refer to the Canvas page Term Project | MongoDB Categorizations/Inheritance for guidance regarding modeling/implementing categorizations in Moon Modeler and MongoEngine.
      2. There are three ways to physically implement a categorization. We will go over those in class. You can use any one of those in your MongoDB design.
   8. Do **not** worry about CourseCompletion and Incomplete categories of Enrollment.
2. Identification (AKA uniqueness constraints)
   1. In General
      1. MongoDB has no concept of a "primary key".
      2. When a document references another document, you are free to use all the attributes of any one of the uniqueness constraints in the referenc**ed** document's collection. For instance, courses could reference the department's name, or the abbreviation. Or even both if you find that useful.
      3. You are free to "migrate" the "\_id" attribute from the "parent" to the "child" if you like. It is a surrogate, and therefore not descriptive, but it is unique. Moon Modeler facilitates this sort of “foreign key migration” as does MongoEngine. You are free to “migrate” additional/other fields to support your use of the data.
   2. Department
      1. {name}
      2. {abbreviation}
      3. {chairName}
      4. {building, office}
   3. Course
      1. {departmentAbbreviation, courseNumber} – Note that you can use departmentName instead of departmentAbbreviation if you choose to "migrate" that into Course.
      2. {departmentAbbreviation, courseName}
   4. Section
      1. {course, sectionNumber, semester, sectionYear} – Note that course here represents whatever attributes you migrate from Course into Section.
      2. {semester, sectionYear, building, room, schedule, startTime}
      3. {semester, sectionYear, schedule, startTime, instructor}
   5. Student
      1. {lastName, firstName} – even though we all know this combination is hardly useful as a uniqueness constraint in the real world.
      2. {eMail}
      3. {\_id} – note, you don't have to worry about creating and managing a separate student\_id since MongoDB will give you one whether you ask for it or not.
   6. Major
      1. {name}
   7. StudentMajor
      1. {*student*, majorName} – Just which attributes go into this uniqueness constraint will depend upon what you "migrate" in from Student.
   8. Enrollment
      1. {*student, section*} – The exact attributes in this uniqueness constraint will depend upon what migrates in to uniquely identify the Student instance who is enrolled and the Section instance that they enrolled in.
      2. {semester, sectionYear, departmentAbbreviation, courseNumber, studentID}
         1. This one is new. Essentially, we are making sure that no student can be enrolled in more than one section of the same course during the same semester.
         2. You may or may not "migrate" departmentAbbreviation and courseNumber into Enrollment. If not, you will have to implement this constraint by performing one or more queries in the database.
3. Other constraints
   1. Department
      1. Abbreviation must be six characters or less.
      2. chairName must be 80 characters or less.
      3. ~~building name must be 10 characters or less.~~
         1. It was pointed out to me that the IN constraint that I have on building (see below) means that you do not have to worry about the length of the building name getting out of hand.
      4. description must be 80 characters or less.
      5. Building IN ('ANAC', 'CDC', 'DC', 'ECS', 'EN2', 'EN3', 'EN4', 'EN5', 'ET', 'HSCI', 'NUR', 'VEC')
   2. StudentMajor
      1. declarationDate <= today
   3. Course
      1. units must be no less than 1 and no greater than 5.
      2. courseNumber must be >= 100 and < 700
   4. Section
      1. schedule is IN ('MW', 'TuTh', 'MWF', 'F', 'S')
      2. semester is IN ('Fall', 'Spring', 'Summer I', 'Summer II', 'Summer III', 'Winter')
      3. room > 0 and room < 1000 (assume no ten story buildings on campus)
      4. building is IN – the same list as the one for the department office.
      5. startTime >= 8:00 AM and <= 7:30 PM.
   5. PassFail
      1. applicationDate <= today
   6. LetterGrade
      1. minSatistfactory IN ('A', 'B', 'C')
4. Processing
   1. Not every class from the SQLAlchemy project roadmap will implement to a collection if you have elected to embed any of those classes. But I still want your code to allow the user to access individual objects for each of the classes. For instance, if you embed courses into departments, I still want to be able to add, delete, and list courses.
   2. **For each of the classes in the original SQLAlchemy Project Roadmap**, write code that will allow the user to add new instances of those classes. Depending on your design, an instance of a class might be a document in a collection, or it could be an element in an array within a collection.
      1. Make sure that you consider the uniqueness constraint(s) on the document.
      2. Be sure to let the user know of any limitations on the data as they enter it, such as minimum and maximum lengths, value ranges for integers, …
   3. For each of your classes, write code that will allow the user to delete existing objects.
      1. Make sure that you tell them if the object that they select does not exist.
      2. Prevent them from leaving child object "orphan" by deleting their parent.
   4. For each of your classes, write code that will allow the user to print out the objects in that collection.
      1. Pick something sensible to sort by.
      2. Be sure that each of your classes have a \_\_str\_\_ method to make it easy to display objects of that class.
   5. You do not have to start from scratch each time that you run your application.
      1. Test to see whether each uniqueness constraint exists in MongoDB and create it only if you need to. You can do this the same way that I did in the MongoDB Single Table sample code.
      2. You might want to offer the user the choice whether or not to drop the collections for the term project so that you don’t have to drop them manually in Atlas to get a clean start.
      3. You are free to leave data in your collections from one run of your application to the next. Just remember that I will want to see you demonstrate that you can insert new data into all your collections.
5. Error trapping
   1. Use MongoEngine as much as possible to protect the database from corrupt data. Do not try to prevent the user from inputting bad data, let MongoEngine tell you when something is wrong.
   2. Use Python try/except blocks to catch the exceptions that come back from MongoEngine.
   3. Provide the user with sensible error messages so that they know what to do differently when you re-prompt them for the data.
   4. Be sure to give the user another chance to input the data **inside your add function**.

**WHAT TO TURN IN:**

1. The Python code that your team either wrote or updated.
2. Your demo video in which you demonstrate each of the insert/delete/list features and you demonstrate your implementation of the constraints.
3. Your team's design document. If you did not find a need to change your design, then just submit what you provided for phase 1. If you need to make changes to the design, then submit a revised version of your design document.
4. Your Moon Modeler .dmm file. You created a Moon Modeler model for this project in the design phase. In this phase I want to see you add the constraints (such as the minimum and maximum length of the strings, enumerations on the semester and building, as well as the uniqueness constraints). Even though this is redundant to the code that you are doing in your MongoEngine classes, I want to see it. Please be careful that the constraints are consistent between your Moon Modeler representation of the constraints and your Python code.
5. Your team's collaboration form. You will find that at Canvas | Modules | Course General Information | Collaboration Form.