PyMongo Introduction

- **Core Functionality**: PyMongo is the official Python driver for MongoDB, providing a Pythonic interface to interact with MongoDB databases
- **Design Philosophy**: Maps MongoDB's document-oriented model directly to Python dictionaries and lists
- **Installation Process**: `pip install pymongo` adds the driver to your Python environment
- **Version Compatibility**: Different PyMongo versions align with specific MongoDB server versions
- **Connection Handling**:
    - Uses MongoDB connection string URI format
    - Supports authentication, TLS/SSL encryption, connection pooling
    - Connection string format: `mongodb://[username:password@]host[:port][/database][?options]`
    - Default port is 27017 if not specified
- **Error Handling**: Provides specific exception types for different MongoDB operation failures

MongoDB Connection with PyMongo

- **Connection Establishment**:
    - `MongoClient` is the entry point for all MongoDB operations
    - Can specify connection parameters as URI string or individual parameters
    - Handles connection pooling automatically
    - Default timeout settings can be customized for production environments
- **Connection Scenarios**:
    - Local development usually connects to localhost
    - Production typically requires authentication credentials
    - Replica sets need multiple hosts specified
    - Connection to MongoDB Atlas requires special configuration
- **Security Considerations**:
    - Best practice is to store credentials in environment variables
    - Connection strings with passwords should never be hardcoded
    - TLS/SSL should be enforced in production

Database and Collection Access

- **Database Access Methods**:

    - Dictionary-style access: `db = client['ds4300']`
    - Attribute-style access: `db = client.ds4300`
    - Creates database on first document insertion if it doesn't exist
- **Collection Access Methods**:
    - Dictionary-style: `collection = db['myCollection']`
    - Attribute-style: `collection = db.myCollection`
    - Collections are created lazily upon first insertion
- **Database Operations**:
    - List databases: `client.list_database_names()`
    - Drop database: `client.drop_database('db_name')`
- **Collection Operations**:
    - List collections: `db.list_collection_names()`
    - Create collection with options: `db.create_collection(name, options)`
    - Drop collection: `db.drop_collection('collection_name')`

Document Operations

- **Inserting Documents**:
    - `insert_one()`: Inserts a single document and returns `InsertOneResult` object
    - Document ID is auto-generated if not provided (ObjectId type)
    - Inserted document is modified with `_id` field if it wasn't present
    - Returns the inserted document's ID via `inserted_id` property
- **Insert Performance Considerations**:
    - Single-document operations provide strong consistency guarantees
    - Batch operations with `insert_many()` offer better performance for multiple documents
    - ObjectIds are designed to be generated efficiently without coordination
    - Ordered inserts (default) stop on first error; unordered continue despite errors
- **Document Structure**:
    - MongoDB documents are represented as Python dictionaries
    - Can contain nested dictionaries and lists
    - Special BSON types (ObjectId, Decimal128, etc.) available in `bson` package
    - Field names have restrictions (no dots, dollar signs at start)

Querying Documents

- **Basic Querying**:
    - `find()` returns a cursor to matching documents
    - `find_one()` returns a single document or None
    - First parameter is a query document specifying criteria

- **Query Operators**:
  - Comparison: `$eq`, `$gt`, `$gte`, `$lt`, `$lte`, `$ne`, `$in`, `$nin`
  - Logical: `$and`, `$or`, `$not`, `$nor`
  - Element: `$exists`, `$type`
  - Array: `$all`, `$elemMatch`, `$size`
- **Cursor Methods**:
  - `sort()`: Controls order of results
  - `limit()`: Restricts number of results
  - `skip()`: Skips initial results
  - `count_documents()`: Counts matching documents
  - `distinct()`: Returns unique values for a field
- **Handling Results**:
  - Cursor is iterable but can only be used once
  - `bson.json_util.dumps()` converts BSON to JSON-compatible format
  - Pretty-printing with `indent` parameter for debugging

Jupyter Notebook Integration

- **Environment Setup**:
  - Separate conda/virtualenv environments isolate dependencies
  - Installing all required packages in the same environment ensures compatibility
  - JupyterLab offers more features than classic Jupyter Notebook
- **Interactive Development**:
  - Jupyter notebooks provide excellent environment for exploring MongoDB data
  - Cell-by-cell execution allows incremental development
  - Rich output formats help visualize query results
  - Magic commands like `%time` useful for performance analysis
- **Sharing and Collaboration**:
  - Notebooks can be shared with query examples
  - Results can be exported to various formats
  - Version control integration possible with extensions
- **Best Practices**:
  - Keep connection strings in separate config cells
  - Use markdown cells to document complex queries
  - Handle large result sets carefully to avoid memory issues
  - Consider using pandas for advanced data analysis of query results

Advanced PyMongo Features (Beyond Slides)

- **Aggregation Framework**:
  - `aggregate()` method supports MongoDB's powerful aggregation pipeline

    - Multiple stages: `$match`, `$group`, `$project`, `$sort`, etc.
    - Allows complex data transformations and analytics
- **Change Streams**:
    - Real-time notifications of database changes
    - Can watch collections, databases, or deployments
    - Enables event-driven architectures
- **Transactions**:
    - Multi-document ACID transactions
    - Uses `with` statement for session management
    - Requires replica sets or sharded clusters
- **GridFS**:
    - System for storing large files (>16MB)
    - Splits files into chunks
    - Provides file-like interface
- **Geospatial Queries**:
    - Support for 2d and 2dsphere indexes
    - Proximity queries with `$near` and `$geoWithin`
    - GeoJSON format for geographic data

Performance and Optimization Techniques

- **Indexing Strategies**:
    - Creating indexes: `create_index()` and `create_indexes()`
    - Index types: single-field, compound, multikey, text, geospatial
    - Index options: unique, sparse, TTL, partial
- **Query Optimization**:
    - Use `explain()` to understand query execution plans
    - Cover queries with appropriate indexes
    - Projection to limit fields returned
    - Batch processing for large datasets
- **Connection Pooling**:
    - PyMongo handles connection pools automatically
    - Configure `maxPoolSize` for high-concurrency applications
    - Monitor pool with MongoDB server stats
- **Bulk Operations**:
    - Bulk write API for efficient multiple operations
    - Ordered vs. unordered execution
    - Reduces network overhead

Deployment Considerations

- **Production Setup**:

    - Security: Authentication, TLS/SSL, network isolation
    - Monitoring: Application metrics and MongoDB server stats
    - High availability: Replica sets configuration
    - Scalability: Sharding for horizontal scaling
- **Error Handling**:
    - Implement robust error handling with appropriate retries
    - Differentiate between transient and permanent errors
    - Timeout settings for long-running operations
- **Logging and Debugging**:
    - Enable PyMongo logging for troubleshooting
    - Monitor slow queries
    - Use database profiler for detailed query analysis