# Chapter-3

**Topic-1: Working with modules and packages in Python:**

**Module:** A module is essentially a straightforward Python file consisting of a grouping of functions and global variables, distinguished by its .py file extension. It serves as an executable file, and to manage all these modules, Python introduces the idea of a Package.
**Example:** Random

**Package:** The package consists of a straightforward directory with groups of modules. In addition to Python modules, this directory includes a __init__.py file that the interpreter uses to interpret it as a package. All that the package is is a namespace. Sub-packages are also included in the package.
**Example:** Numpy, Pandas

**Library:** A Python library is a group of codes or code modules that can be used for particular tasks in a program.

| Aspect | Module | Package | Library |
|---|---|---|---|
| **Definition** | A single Python file (.py) | A directory of related modules | A collection of modules and packages |
| **Organization** | Single file | Folder with __init__.py file | May contain multiple packages and modules |
| **Scope** | Smallest unit for reuse | Medium-level organization | Broader collection of utilities |
| **Example** | math, os, my_module.py | MyPackage (folder with modules) | NumPy, Pandas, requests |

**Topic-2: Difference between relational and NoSQL database:**

**Relational Databases**

- Data is organized into tables with rows and columns, forming a structured schema.
- Based on the relational model, using SQL (Structured Query Language) for data manipulation.
- Define relationships between different tables (e.g., one-to-one, one-to-many) using foreign keys.
- Strong data integrity and consistency.
- Well-suited for complex queries and transactions.
- Mature technology with robust tools and support.
- Can be less flexible for rapidly evolving data structures.
- Scalability can be challenging with very large datasets.
- May not be ideal for unstructured or semi-structured data.

**Non-Relational Databases (NoSQL)**

- Use various data models like key-value pairs, document stores, column families, and graph databases.
- More flexible and schema-less or schemaless, allowing for dynamic data structures.
- Relationships between data are often implicit or handled differently depending on the data model.
- Highly scalable and can handle massive volumes of data.
- Excellent for handling unstructured or semi-structured data.

- More flexible for rapidly changing data requirements.
- May have weaker data consistency guarantees compared to relational databases.
- Can be more complex to query and manage in some cases.

**Topic-3: Python Connectivity with MySQL:**

**Mysql-connector:**

- **MySQL Connector/Python** is a library that enables Python programs to connect and interact with MySQL databases.
- **Connection Establishment:** Provides methods to establish a connection to a MySQL server, specifying credentials (user, password, host) and database name.
- **SQL Execution:** Allows you to execute various SQL statements (SELECT, INSERT, UPDATE, DELETE) through the connection.
- **Data Retrieval:** Facilitates fetching data from the database using cursors and methods like fetchone(), fetchall(), fetchmany().
- **Error Handling:** Provides mechanisms for handling potential errors during database operations (e.g., connection errors, SQL execution errors).
- **Data Types:** Supports conversion between Python data types and MySQL data types.

**Cursor():**

- The cursor navigates through a query's result set like a pointer.
- By enabling you to run SQL commands and access or alter data, it offers an organised method of interacting with the database.
- Cursors are used to execute SQL queries against the database.
- They provide methods like execute() to send SQL statements to the database for processing.

**fetchone()**

- Retrieves only the next single row from the result set.
- Used when you need to process data row by row.
- Used when you have a large dataset and want to minimize memory usage.
- Used in scenarios where you don't need all the data at once.

**fetchall()**

- Retrieves all the rows from the result set at once.
- Used when you need to process the entire dataset in memory.
- Used when the dataset is relatively small.

**Example:**

```
import mysql.connector

# Connect to the database
mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="mydatabase"
)
```

```
# Create a cursor object
mycursor = mydb.cursor()

# Execute a SQL query
mycursor.execute("SELECT * FROM customers")

# Fetch all rows
results = mycursor.fetchall()

# Print the results
for row in results:
  print(row)

# Close the connection
mydb.close()
```

**Topic-4: Python connectivity with MongoDB:**

- The **pymongo** library is the official Python driver for MongoDB. It provides a comprehensive interface for interacting with MongoDB databases from within your Python applications.
- PyMongo acts as a bridge between your Python code and MongoDB, enabling seamless interaction with your MongoDB database.
- It provides a powerful and flexible API for performing a wide range of database operations, making it an essential tool for developing MongoDB-driven applications in Python.
- PyMongo enables you to access and interact with specific databases within your MongoDB instance.
- PyMongo allows you to create new databases and collections as needed, and also drop them when necessary.

**Example:**

```
from pymongo import MongoClient

# Connect to the default MongoDB instance (localhost:27017)
client = MongoClient()

# Connect to a specific host and port
# client = MongoClient('localhost', 27017)

# Connect with authentication (if required)
# client = MongoClient('mongodb://<username>:<password>@<host>:<port>/')
db = client['your_database_name']
collection = db['your_collection_name']
data = {'name': 'John Doe', 'age': 30, 'city': 'New York'}
result = collection.insert_one(data)
print(result.inserted_id)
# Find all documents
results = collection.find()

# Find documents with specific criteria
```

```
results = collection.find({'age': {'$gt': 25}})

for doc in results:
    print(doc)
client.close()
```