# UNIT-5
# Database Programming

**Persistent Storage:**

Persistent storage is extremely vital in computer systems. This is because if all data was volatile we wouldn't be able permanently keep data for later use, as it would all be gone once we turned off the system.

Persistent storage is necessary in order to be able to keep all our files and data for later use. For example a hard disk drive is an example of persistent storage; it allows us to permanently store a variety of data.

Persistent storage refers to non-volatile storage. Persistent storage is needed to store data in a non-volatile device during and after the running of a program. Be careful to clearly understand persistent storage is required not only when your computer is turned off, but also when data needs to be stored due to the end of a computer process or lack of systems resources.

To be specific persistent storage is needed when:

- The system is powered off.
- Another process requires data / results from volatile memory (RAM, loses content when power is off.)
- There is a need to store data during and after the running(processing done in CPU which has no storage) of a program.

In any application, there is a need for persistent storage. Generally, there are three basic storage mechanisms:

1. Files
2. A relational database system (RDBMS)
3. An API (application programmer interface) that "sits on top of" one of those existing systems an object relational mapper (ORM), file manager, spreadsheet, configuration file, etc.

**Database Programming Introduction:**

- To build the real world applications, connecting with the databases is the necessity for the programming languages.

- Python allows us to connect our application to the databases like MySQL, SQLite, MongoDB, and many others.

- Python also supports Data Definition Language (DDL), Data Manipulation Language (DML) and Data Query Statements. For data base programming, the Python DB-API is a widely used module that provides a database application programming interface.

The Python Programming language has powerful features for database programming, those are

- Python is famous for its portability.
- It is platform independent.
- In many programming languages, the application developer needs to take care of the open and closed connections of the database, to avoid further exceptions and errors. In Python, these connections are taken care of.
- Python supports relational database systems.
- Python database APIs are compatible with various databases, so it is very easy to migrate and port database application interfaces.

**Database concepts and Structured Query Language (SQL)**

**Underlying Storage**

Databases usually have a fundamental persistent storage using the file system, i.e., normal operating system files, special operating system files, and even raw disk partitions.

**User Interface**

Most database systems provide a command-line tool with which to issue SQL commands or queries. There are also some GUI tools that use the command-line clients or the database client library, giving users a much nicer interface.

**Databases**

An RDBMS can usually manage multiple databases, e.g., sales, marketing, customer support, etc., all on the same server (if the **RDBMS** is server-based; simpler systems are usually not). In the examples we will look at in this chapter, **MySQL** is an example of a server-based RDBMS because neither there is a server process running continuously waiting for commands while neither SQLite nor Gadfly have running servers.

**Components**

The **table** is the storage abstraction for databases. Each row of data will have fields that correspond to database columns. The set of table definitions of columns and data types per table all put together define the database schema.

Databases are created and dropped. The same is true for tables. Adding new rows to a database is called inserting, changing existing rows in a table is called updating, and removing existing rows in a table is called deleting. These actions are usually referred to as database commands or operations.

Requesting rows from a database with optional criteria is called querying. When you query a database, you can fetch all of the results (rows) at once, or just iterate slowly over each resulting row. Some databases use the concept of a cursor for issuing SQL commands, queries, and grabbing results, either all at once or one row at a time.

**SQL**

Database commands and queries are given to a database by SQL. Not all databases use SQL, but the majority of relational databases do. Here are some examples of SQL commands. Most databases are configured to be case-insensitive, especially database commands. The accepted style is to use CAPS for database keywords. Most command-line programs require a trailing semicolon ( ; ) to terminate a SQL statement.

**Create, use and drop the database using SQL command:**

**Creating a database and granting the permission to the user:** To perform operations on database, database must create first and all permission must be granted to the user.

The following create database command is used to creates a database named "test":

```
CREATE DATABASE test;
```

Grant permissions to specific users (or all of them) so that they can perform the database operations on the data base test. The GRANT command is as follows:

```
GRANT ALL ON test.* to user(s);
```

If you logged into a database system without choosing which database you want to use, USE SQL statement allows you to specify one with which to perform database operations.

```
USE test;
it allows to perform database operations on test database.
```

DROP statement removes all the tables and data from the database and deletes it from the system.

```
DROP DATABASE test;
It removes the test database from the system.
```

**Create and drop a table in SQL:**

Creating a Table: CREATE statement creates a new table with a string column login and a pair of integer field's uid and prid.

```
CREATE TABLE users (login VARCHAR(8), uid INT, prid INT);
```

Dropping a Table: DROP statement drops a database table along with all its data.

```
DROP TABLE users;
```

**Insert, update and delete records**

**Insert a Row:** A new row can be inserted in a database with the INSERT statement. Specify the table and the values that go into each field.

**Ex:**

```
INSERT INTO users VALUES ('leanna', 311, 1);
```

The string 'leanna' goes into the login field, and 311 and 1 to uid and prid, respectively.

**Update a Row:** We can use the UPDATE statement to change existing table rows. Use SET for the columns that are changing and provide any criteria for determining which rows should change.

**Ex:**

```
UPDATE users SET prid=4 WHERE prid=2;
```

all users with a "project ID" or prid of 2 will be moved to project #4.

```
UPDATE users SET prid=1 WHERE uid=311;
```

We take one user (with a UID of 311) and move them to project #1.

**Delete a Row:** Use the DELETE FROM command to delete a table row, give the table you want to delete rows from, and any optional criteria.

Ex:

```
DELETE FROM users WHERE prid=%d;
```

```
DELETE FROM users;
```

All rows will be deleted.

**Query the database:** Select statement is used to retrieve the rows from the database table.

Ex:

```
SELECT * from users;
```

It retrieves all the rows from table users.

```
SELECT * from users where uid = 311;
```

It retrieves all the fields of a row whose uid is 311.

**Databases and Python:**

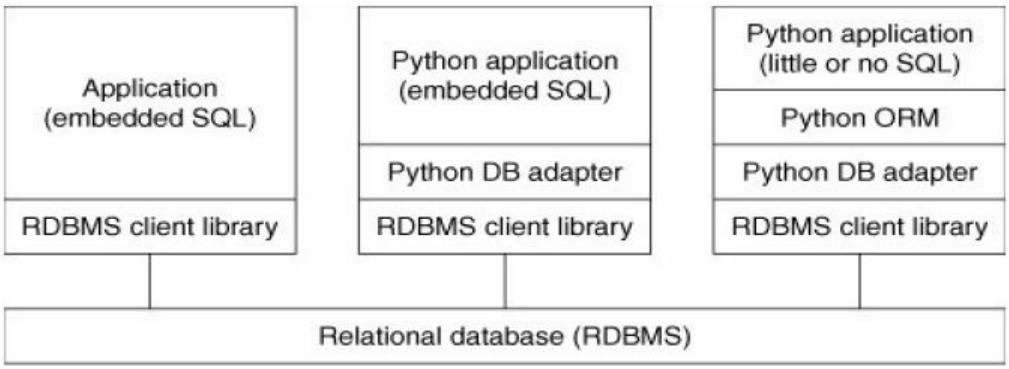
We will present how to store and retrieve data to/from RDBMSs while playing within a Python framework.

The goal is to get you up to speed as quickly as possible if you need to integrate your Python application with some sort of database system.

it has become clear that being able to work with databases is really a core component of everyday application development in the Python world.

As a software engineer, you can probably only make it so far in your career without having to learn something about databases: how to use one (command-line and/or GUI interfaces), how to pull data out of one using the Structured Query Language (SQL), perhaps how to add or update information in a database, etc. If Python is your programming tool, then a lot of the hard work has already been done for you as you add database access to your Python universe. We first describe what the Python "DB-API" is, then give examples of database interfaces that conform to this standard.

The way to access a database from Python is via an adapter. An adapter is basically a Python module that allows you to interface to a relational database's client library, usually in C. It is recommended that all Python adapters conform to the Python DB-SIG's Application Programmer Interface (API). This is the first major topic of this chapter.




Multi-tiered Communication between Application and Database

- The first box is generally a C/C++ program
- The second box, DB-API compliant adapters let you program applications in Python.
- The third box, ORMs can simplify an application by handling all of the database-specific details.

**Python Database Application Programmer's Interface (DB-API)**

- The API is a specification that states a set of required objects and database access mechanisms to provide consistent access across the various database adapters and underlying database systems.
- The API was driven by strong need. In the "old days," we had a scenario of many databases and many people implementing their own database adapters. It was a wheel that was being reinvented over and over again.
- These databases and adapters were implemented at different times by different people without any consistency of functionality. Unfortunately, this meant that application code using such interfaces also had to be customized to which database module they chose to use, and any changes to that interface also meant updates were needed in the application code.
- A special interest group (SIG) for Python database connectivity was formed, and eventually, an API was born ... the DB-API version 1.0. The API provides for a consistent interface to a variety of relational databases, and porting code between different databases is much simpler.

**DB-API Module Attributes:**

The DB-API specification mandates that the features and attributes listed below must be supplied. A DB- API-compliant module must define the global attributes as shown below.

| **Attribute** | **Description** |
|---|---|
| apilevel | Version of DB-API module is compliant with |
| threadsafety | Level of thread safety of this module |
| paramstyle | SQL statement parameter style of this module |
| Connect() | Connect() function |

**Data Attribute:**

**apilevel:** This string (not float) indicates the highest version of the DB-API the module is compliant with, i.e., "1.0", "2.0", etc. If absent, "1.0" should be assumed as the default value.

**threadsafety:** This an integer with these possible values:

- 0: Not threadsafe, so threads should not share the module at all
- 1: Minimally threadsafe: threads can share the module but not connections
- 2: Moderately threadsafe: threads can share the module and connections but not cursors.
- 3: Fully threadsafe: threads can share the module, connections, and cursors

**paramstyle:**

- The API supports a variety of ways to indicate how parameters should be integrated into an SQL statement that is eventually sent to the server for execution. This argument is just a string that specifies the form of string substitution you will use when building rows for a query or command.

| Parameter Style | Description | Example |
|---|---|---|
| numeric | Numeric positional style | WHERE name=:1 |
| named | Named style | WHERE name=:name |
| pyformat | Python dictionary printf() format conversion | WHERE name=%(name)s |
| qmark | Question mark style | WHERE name=? |
| format | ANSI C printf() format conversion | WHERE name=%s |

**Function Attribute(s):**

- connect() Function access to the database is made available through Connection objects.
- A compliant module has to implement a connect() function, which creates and returns a Connection object.
- Attributes or parameters of connect () function are describes as follows:

| Parameter | Description |
|---|---|
| user | Username |
| password | Password |
| host | Hostname |
| database | Database name |
| dsn | Data source name |

- You can pass in database connection information as a string with multiple parameters (DSN) or individual parameters passed as positional arguments (if you know the exact order).
    - Here is an example of using connect() from PEP 249:
      connect(dsn='myhost:MYDB',user='guido',password='234$')

**DB-API Exception Classes:**

- There are many sources of errors, syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.
- The DB-API defines a number of errors that must exist in each database module.
- Exception classes and their description is as follows:

| Exception | Description |
|---|---|
| Warning | Root warning exception class |
| Error | Root error exception class |
| InterfaceError | Database interface (not database) error |
| DatabaseError | Database error |
| DataError | Problems with the processed data |
| OperationalError | Error during database operation execution |
| IntegrityError | Database relational integrity error |
| InternalError | Error that occurs within the database |
| ProgrammingError | SQL command failed |
| NotSupportedError | Unsupported operation occurred |

**Connection Object and its methods:**

- Connections are how your application gets to talk to the database. They represent the fundamental communication mechanism by which commands are sent to the server and results returned.

- Once a connection has been established (or a pool of connections), you create cursors to send requests to and receive replies from the database.

- Connection objects are not required to have any data attributes but should define the methods, the methods which can be called on connection objects are as follows:

| Method Name | Description |
|---|---|
| close() | Close database connection |
| commit() | Commit current transaction |
| rollback() | Cancel current transaction |
| cursor() | Create (and return) a cursor or cursor-like object using this connection |

errorhandler(cxn, cur, errcls, errval) Serves as a handler for given connection cursor

When close() is used, the same connection cannot be used again without running into an exception.

The commit() method is irrelevant if the database does not support transactions or if it has an auto-commit feature that has been enabled.

Like commit(), rollback() only makes sense if transactions are supported in the database. After execution, rollback() should leave the database in the same state as it was when the transaction began.

If the RDBMS does not support cursors, cursor() should still return an object that faithfully emulates or imitates a real cursor object. These are just the minimum requirements.

**Cursor Object and its methods:**

- Once you have a connection, you can start talking to the database.
- A cursor lets a user issue database commands and retrieve rows resulting from queries.
- It is an object that is used to make the connection for executing SQL queries. It acts as middleware between database connection and SQL query. It is created after giving connection to database.
- You can create Cursor object using the cursor() method of the Connection object/class.
- Once you have created a cursor, you can execute a query or command (or multiple queries and commands) and retrieve one or more rows from the results set.
- A Python DB-API cursor object functions as a cursor for you, even if cursors are not supported in the database. In this case, the database adapter creator must implement CURSOR objects so that they act like cursors.
- This keeps your Python code consistent when you switch between database systems that have or do not have cursor support.

**The cursor object attributes and their description is as follows:**

| Object Attribute | Description |
|---|---|
| arraysize | Number of rows to fetch at a time with fetchmany(); defaults to 1 |
| connection | Connection that created this cursor (optional) |
| description | Returns cursor activity (7-item tuples): (name, type_code, display_size, internal_ size, precision, scale, null_ok); only name and type_code are required |
| lastrowid | Row ID of last modified row (optional; if row IDs not supported, default to None) |
| rowcount | Number of rows that the last execute*() produced or affected |
| rownumber | Index of cursor (by row, 0-based) in current result set (optional) |

**The cursor object methods and their description is as follows:**

| Object Method | Description |
|---|---|
| callproc(func[, args]) | Call a stored procedure |
| close() | Close cursor |
| execute(op[, args]) | Execute a database query or command |
| executemany(op, args) | Like execute() and map() combined; prepare and execute a database query or command over given argument. |
| fetchone() | Fetch next row of query result. |

fetchmany ([size=cursor.arraysize]) Fetch next size rows of query result

| | |
|---|---|
| fetchall() | Fetch all (remaining) rows of a query result |
| __iter__() | Create iterator object from this cursor (optional; also see next()) messages List of messages (set of tuples) received from the database for cursor execution (optional) |
| next() | Used by iterator to fetch next row of query result (optional; like fetchone(), also see __iter__()) |

| | |
|---|---|
| nextset() | Move to next results set (if supported) |
| setinputsizes(sizes) | Set maximum input size allowed (required but implementation optional) |
| setoutputsize(size[,col]) | Set maximum buffer size for large column fetches (required but implementation optional) |

**Type Objects and Constructors**

- The interface between two different systems is the most fragile. This is seen when converting Python objects to C types and vice versa.
- Similarly, there is also a fine line between python objects and native database objects.
- As a programmer writing to Python's DB-API, the parameters you send to a database are given as strings, but the database may need to convert it to a variety of different, supported data types that are correct for any particular query.
- Example: should the Python string be converted to a VARCHAR, a TEXT, a BLOB, or a raw BINARY object.
- Another requirement of the DB-API is to create constructors that build special objects that can easily be converted to the appropriate database objects.

**Constructor:** a constructor is a special method which is used to create an object. A constructor is called when an object is created. The constructor is defined as member inside the definition of a class. It can also be used to initialize the object attributes.

Python DB-API is to create constructors that build special objects that can easily be converted to the appropriate database objects.

| **Type Object** | **Description** |
|---|---|
| Date(yr, mo, dy) | Object for a date value |
| Time(hr, min, sec) | Object for a time value |
| Timestamp(yr, mo, dy, hr, min, sec) | Object for a timestamp value |
| DateFromTicks(ticks) | Date object given number of seconds since the epoch |
| TimeFromTicks(ticks) | Time object given number of seconds since the epoch |
| TimestampFromTicks(ticks) | Timestamp object given number of seconds since the epoch |

| | |
|---|---|
| Binary(string) | Object for a binary (long) string value |
| STRING | Object describing string-based columns, e.g., VARCHAR |
| BINARY | Object describing (long) binary columns, i.e., RAW, BLOB |
| NUMBER | Object describing numeric columns |
| DATETIME | Object describing date/time columns |
| ROWID | Object describing "row ID" columns |

**Steps to create a database connection using python DB-API:**

- PythonDB-API is independent of any database engine, which enables you to write Python scripts to access any database engine.
- The PythonDB-API implementation for MySQL is possible by MySQLdb or mysql.connector.
- Using Python structure, DB-API provides standard and support for working with databases.

The API consists of:

1. Import module (mysql.connectororMySQLdb)
2. Create the connection object.
3. Create the cursor object
4. Execute the query
5. Close the connection

1. **Import module (mysql.connectoror MySQLdb):**
   To interact with MySQL database using Python, you need first to import mysql.connector or MySQLdb module by using following statement.

MySQLdb (in python2.x)

```
import MySQLdb
```

mysql.connector(in python3.x)

```
import mysql.connector
```

**2. Create the connection object:**

After importing mysql.connector or MySQLdb module, we need to create connection object, for that pythonDB-API supports one method i.e. connect()method.

It creates connection between MySQL database and Python Application.

If you import MySQLdb (inpython2.x) then we need to use following code to create connection.

Syntax:

Conn-name = MySQLdb.connect(<hostname>,<username>,<password>,<database>)

Example:

Myconn=MySQLdb.connect("localhost","root","root",”emp”)

(Or)

If you import mysql.connector (inpython3.x) then we need to use following code to create connection.

Syntax:

conn-name = mysql.connector.connect(host=<host-name>, user=<username>, passwd=<pwd>, database=<dbname>)

Example:

myconn=mysql.connector.connect(host="localhost",user="root", passwd="root", database=”emp”)

**3. Create the cursor object:**

After creation of connection object we need to create cursorobject to execute SQL queries in MySQL database.

The cursor object facilitates us to have multiple separate working environments through the same connection to the database.

The Cursor object can be created by using cursor() method.

Syntax:

cur_came = conn-name.cursor()

Example:

my_cur = myconn.cursor()

4. **Execute the query:**

   After creation of cursor object we need to execute required queries by using cursor object.

   To execute SQL queries, pythonDB-API supports following method i.e. execute().

   Syntax:

   cur-name.execute(query)

   Example:

   my_cur.execute("select * from Employee")

5. **Close the connection:**

   After completion of all required queries we need to close the connection.

   Syntax:

   conn-name.close()

   Example:

   conn-name.close()

**Write a python script to create a table, insert and display data from the database.**

```
#!C:/Users/DELL/AppData/Local/Programs/Python/Python310/pythonw.exe
print("Content-Type:text/html")
print()

import cgi
import mysql.connector

# creating database connection object
conn_obj  = mysql.connector.connect(host="localhost",user="root",password="",database="stu_db")

# creating cursor object
cursor = conn_obj.cursor()
```

```
# creating a table
cursor.execute("CREATE TABLE stu_fee(roll_no VARCHAR(15),name VARCHAR(15),fee FLOAT)")

# inserting records
cursor.execute("INSERT INTO stu_fee VALUES('21QM1A0401','Nitheesha',75000)")
cursor.execute("INSERT INTO stu_fee VALUES('21QM1A0402',Ranjith',75000)")

conn_obj.commit()

#Retrieve and display data from database table.
cursor.execute("SELECT * from stu_fee")

for row in cursor:
    print(row)

# closing connection
conn_obj.close()
```

**Write a python script to store and retrieve student details from database through web programming:**

**Stu_fee.html**

```
html>
<head>
<title> Student Form</title>
</head>

<body>

<form method="GET" action="stu_fee.py">
Roll Number: <input type=text name=stu_rno value=" ">
Studet Name: <input type=text name=stu_name value=" ">
Student Fee: <input type=text name=stu_fee value=" ">
<input type = submit>
</form>

</body>

</html>
```

**Stu_fee.py**

```
#!C:/Users/DELL/AppData/Local/Programs/Python/Python310/pythonw.exe
print("Content-Type:text/html")
print()

import cgi
import mysql.connector

form = cgi.FieldStorage()

rno = form.getvalue("stu_rno")

name = form.getvalue("stu_name")

fee = float(form.getvalue("stu_fee"))

conn_obj = mysql.connector.connect(host="localhost",user="root",password="",database="stu_db")

cursor = conn_obj.cursor()
cursor.execute("insert into stu_fee values(%s,%s,%f)",(rno,name,fee))

conn_obj.commit()

cursor.execute("SELECT * FROM stu_fee")

for row in cursor:
    print(row)

conn_obj.close()
```

**Object Relation Managers:**

- Object Relational Mapping is a system of mapping objects to a database.
- An object-relational mapper (ORM) is a code library that automates the transfer of data stored in relational database tables into objects that are more commonly used in application code.
- ORMs provide a high-level abstraction upon a relational database that allows a developer to write Python code instead of SQL to create, read, update and delete data and schemas in their database.

- Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures.

- The ability to write Python code instead of SQL can speed up web application development, especially at the beginning of a project.

- The potential development speed boost comes from not having to switch from Python code into writing declarative paradigm SQL statements. While some software developers may not mind switching back and forth between languages, it's typically easier to knock out a prototype or start a web application using a single programming language.

- The most well-known Python ORMs today are SQLAlchemy and SQLObject.

- Some other Python ORMs include PyDO / PyDO2, PDO, Dejavu, PDO, Durus, QLime, and ForgetSQL.

- Larger Web-based systems can also have their own ORM component, i.e., WebWare MiddleKit and Django's Database API.

**SQLAlchemy:**

- SQLAlchemy is a library used to interact with a wide variety of databases. It enables you to create data models and queries in a manner that feels like normal Python classes and statements.
- It can be used to connect to most common databases such as Postgres, MySQL, SQLite, Oracle and many others.
- SQLAlchemy is a popular SQL tool kit and Object Relational Mapper. It is written in Python and gives full power and flexibility of SQL to an application developer.
- It is necessary to install SQLAlchemy. To install we have to use following code at Terminal or CMD.

  pip install sqlalchemy

- To check if SQLAlchemyis properly installed or not, enter the following command in the Python prompt

  >>>import sqlalchemy

- If the above statement was executed with no errors, “sqlalchemy" is installed and ready to be used.

**Connecting to Database:**

- To connect with database using SQLAlchemy, we have to create engine for this purpose SQLAlchemy supports one function I screate_engine().

- The create_engine() function is used to create engine; it takes overall responsibilities of database connectivity.

  Syntax:

  Database-server[+driver]://user:password@host/dbname

  Example:

  mysql+mysqldb://root:root@localhost/collegedb

- The main objective of the ORM-API of SQLAlchemy is to facilitate associating user-defined Python classes with database tables, and objects of those classes with rows in their corresponding tables.

**Declare Mapping:**

- First of all, create_engine() function is called to set up an engine object which is subsequently used to perform SQL operations.

- To create engine in case of MySQL:

  Example:

```
from sqlalchemy import create_engine
engine = create_engine('mysql+mysqldb://root:@localhost/Collegedb')
```

- When using ORM, we first configure database tables that we will be using. Then we define classes that will be mapped to them. Modern SQLAlchemy uses Declarative system to do these tasks.

- A declarative base class is created, which maintains a catalog of classes and tables. A declarative base class is created with the declarative_base() function.

- The declarative_base() function is used to create base class. This function is defined in sqlalchemy.ext.declarative module.

- To create declarative base class:

  Example:

```
from sqlalchemy.ext.declarativeimport declarative_base

Base = declarative_base()
```

**tabledef.py**

```
from sqlalchemyimport Column, Integer, String
from sqlalchemyimport create_engine
from sqlalchemy.ext.declarativeimport declarative_base

# create a engine
engine =create_engine('mysql+mysqldb://root:@localhost/Sampledb‘)

# create a declarative base class
Base = declarative_base()
class Students(Base):
__tablename__ = 'students'
id = Column(Integer, primary_key=True)
name = Column(String(10))
address = Column(String(10))
email = Column(String(10))
Base.metadata.create_all(engine)
```

**References:**


1. Core Python Programming, Wesley J. Chun, Second Edition, Pearson.

2. **http://ibdpdang.blogspot.com/2016/11/215-identify-need-for-persistent-storage.html**

3. **https://computersciencewiki.org/index.php/Persistent_storage**

4. **https://www.fullstackpython.com/object-relational-mappers-orms.html#:~:text=An%20object%2Drelational%20mapper%20(ORM,commonly%20used%20in%20application%20code.**