Unit-V
Database Programming

Topics

1. Introduction
2. Python database Application Programmer’s Interface(DB-API)
3. Object-Relational Managers(ORMs)
4. Related Modules

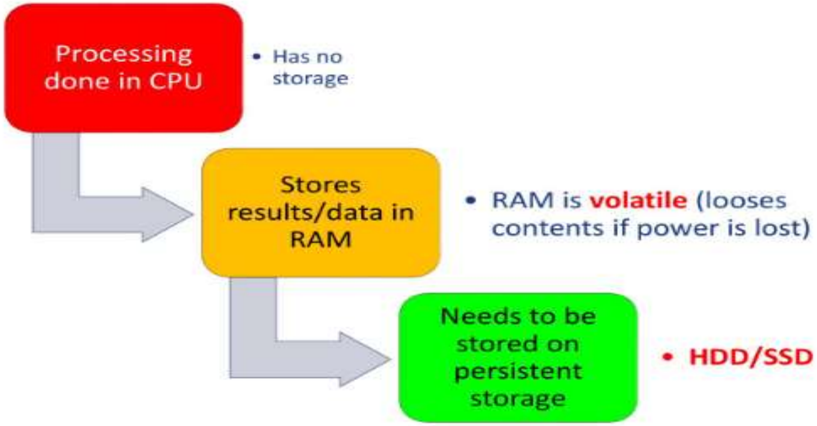
- Storage mechanisms are

1. Files
2. Relational Database Management System (RDBMS)and
3. A type of hybrid i.e., an API (Application Programming Interface) that sits on top of one of those existing systems, file manager, spreadsheet, Object Relational Mapper(ORM), configuration file etc.

1. Introduction

1.1 Persistent Storage

- Persistent storage is any data storage device that preserves data even after power to that device is stopped.
- All applications needs a persistent storage.



Processing done in CPU

- Has no storage

Stores results/data in RAM

- RAM is **volatile** (looses contents if power is lost)

Needs to be stored on persistent storage

- **HDD/SSD**

1.2 Basic Database Operations and SQL

- Some basic database concepts and the Structured Query Language.

1. Underlying Storage:

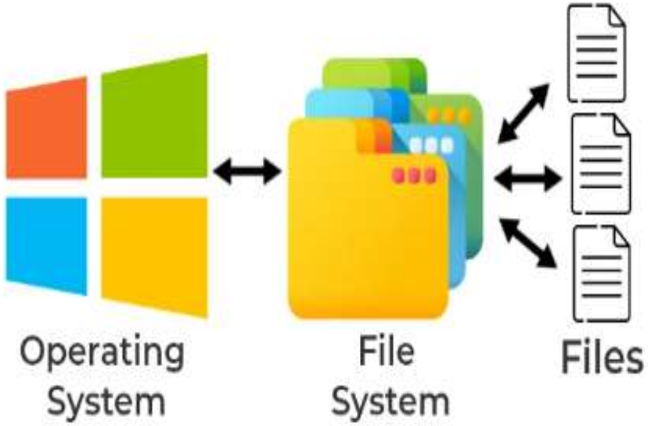
All databases have fundamental persistent storage.

File system –

Normal operating system files

Special operating system files and

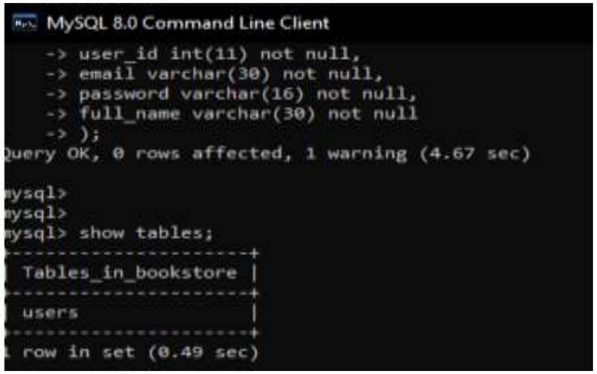
Raw disk partitions



A special file is associated with a particular hardware device or other resource of the computer system.

2. User Interface:

- A command-line tool is used for SQL commands and queries provided by the database systems.

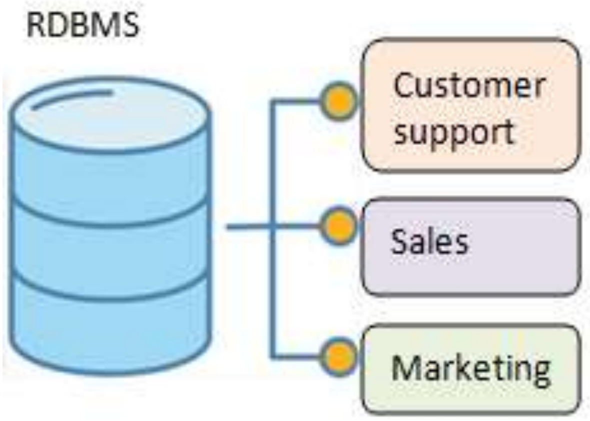


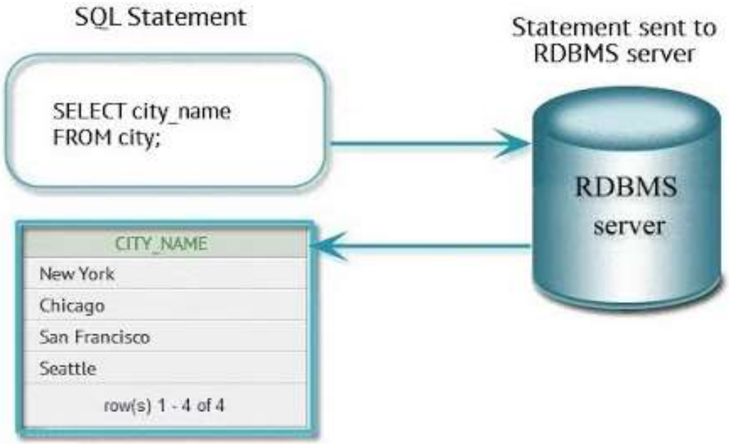
```
    -> user_id int(11) not null,
    -> email varchar(30) not null,
    -> password varchar(16) not null,
    -> full_name varchar(30) not null
    -> );
Query OK, 0 rows affected, 1 warning (4.67 sec)

mysql>
mysql>
mysql> show tables;
+---------------------+
| Tables_in_bookstore |
+---------------------+
| users               |
+---------------------+
1 row in set (0.49 sec)
```

- Graphical User Interface(GUI) tools use command line clients or the database client library, which gives user a nice interface.

3. Databases:

- Database is a systematic collection of data.
- RDBMS has the ability to manage multiple databases such as Customer support, Sales, Marketing etc. all on the same server.
- MySQL is an example of a server-based RDBMS because there is a server process running continuously waiting for commands.





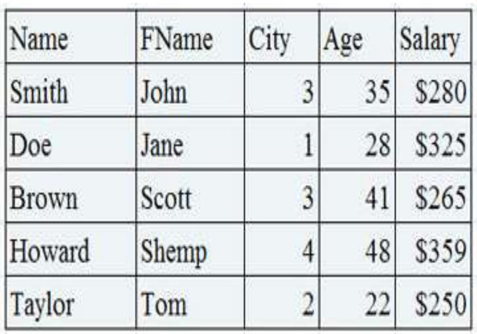
```
SELECT city_name
FROM city;
```

CITY_NAME
New York
Chicago
San Francisco
Seattle

row(s) 1 - 4 of 4

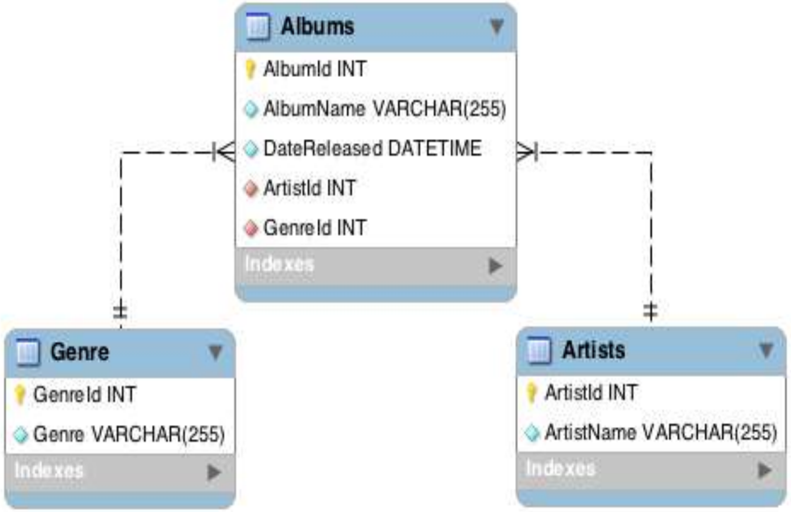
4. Components:

- Table: It is a storage abstraction for databases.
- Each table has rows and columns.



Name	FName	City	Age	Salary
Smith	John	3	35	$280
Doe	Jane	1	28	$325
Brown	Scott	3	41	$265
Howard	Shemp	4	48	$359
Taylor	Tom	2	22	$250

- Database schema: It is a logical representation of a database, which describes how data is stored logically in the database.
- It has a set of table definitions of columns and data types of each table.



- Databases and tables are created and dropped.
- Inserting: adding new rows to a table.
- Updating: changing existing rows in a table.
- Deleting: removing existing rows in a table.
- The above actions are referred to as database commands or operations.
- Querying: requesting rows from a database.
- Query fetches all of the results(rows) at once or just iterate slowly over each resulting row.
- Cursor : it is a concept used for issuing SQL commands, queries, and grabbing results, either all at once or one row at a time.

5. Structured Query Language(SQL):

SQL provides database commands and queries, which are given to a database. Some examples of database commands are:

a. Creating a database:

Example:

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

- It creates a database named test.
- It grants permissions to specific users(or all) to perform the database operations.

b. Using a database:

Example:

```
USE test;
```

- It specifies on which database operations are performed.

c. Dropping a database:

Example:

```
DROP DATABASE test;
```

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

d. Creating a Table:

Example:

```
CREATE TABLE users(userid  INT(4), password VARCHAR(8));
```

- This statement creates a new table with an integer field and a string field.

e. Dropping a table:

Example:

```
DROP TABLE users;
```

- This statement drops a database table along with all its data.

f. Inserting a Row:

Example:

```
INSERT INTO users VALUES(123, ’aaa’);
```

- The integer 123 goes into the userid field, the string ‘aaa’ goes into password field.

g. Updating a Row:

Example:

```
UPDATE users SET userid=456 WHERE userid=123;
```

- This statement sets all userids to 456 from 123.

h. Deleting a Row:

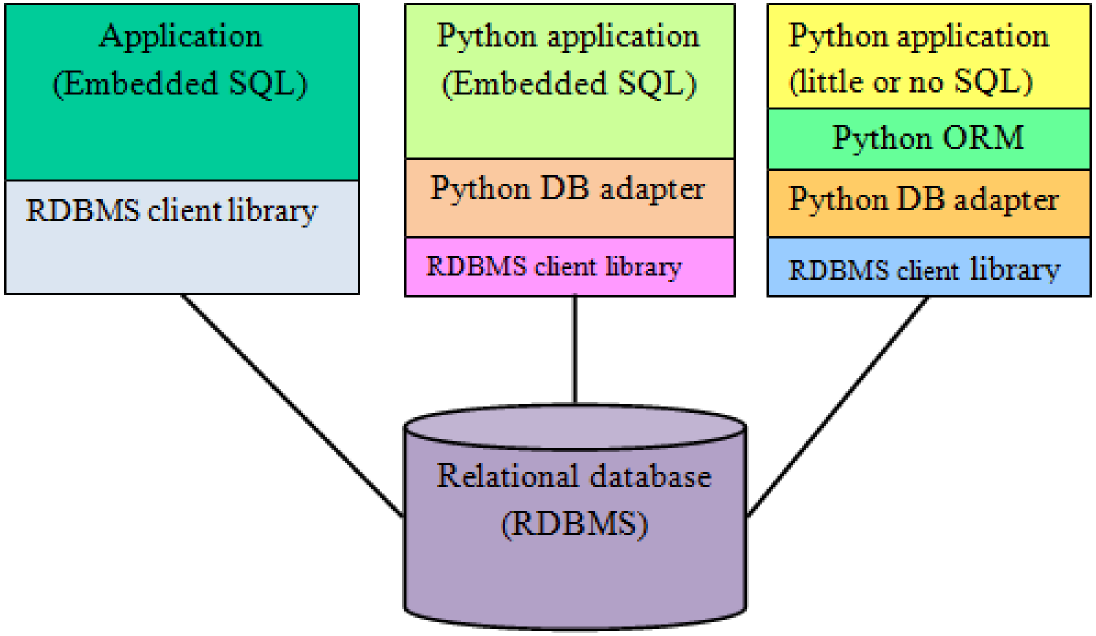
Example:

```
DELETE FROM users WHERE userid=123;
DELETE FROM users;
```

- First line deletes a particular row from users table.
- Second line deletes all rows from users table.

1.3 Databases and Python

- The figure below shows different ways to access database.

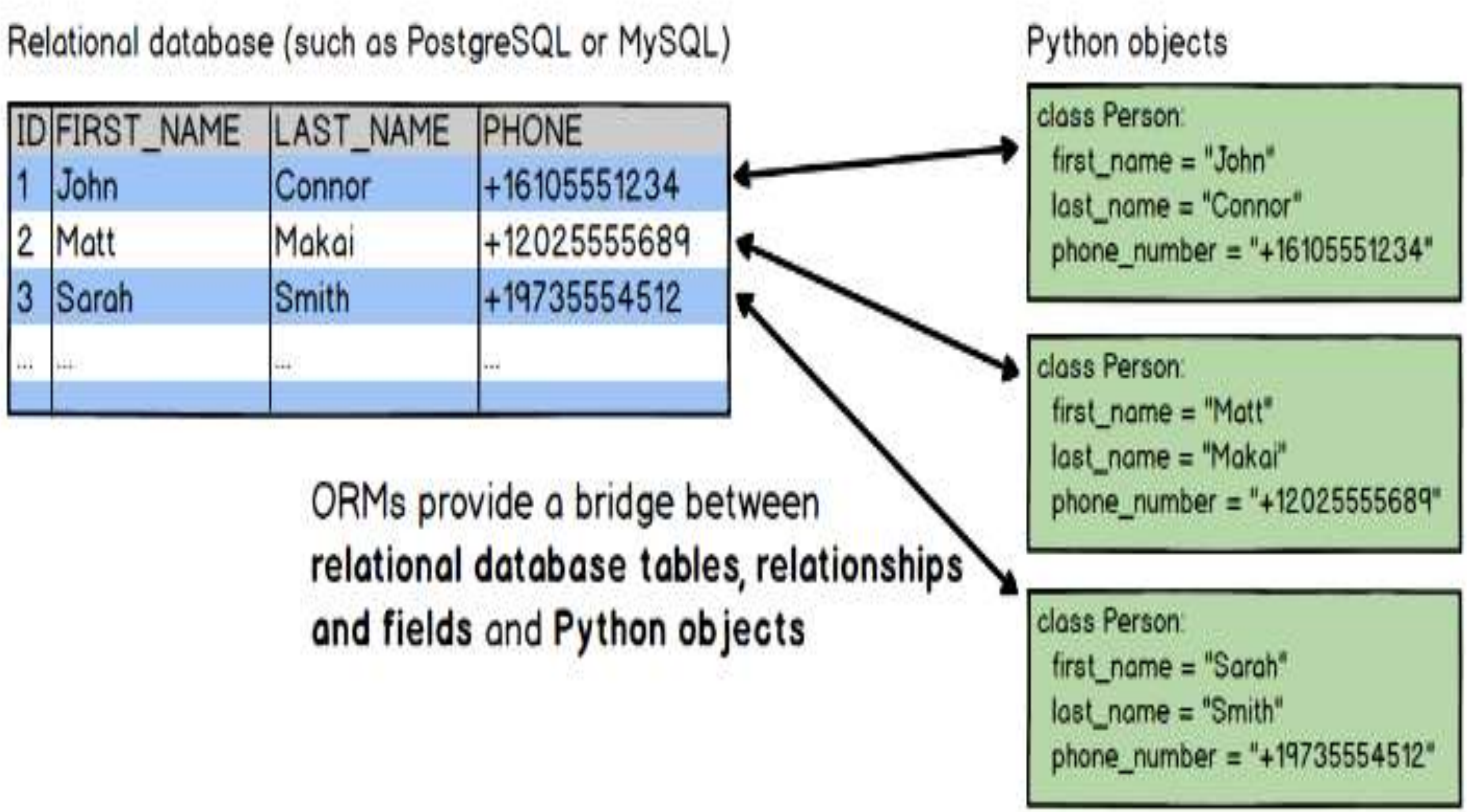


Application (Embedded SQL)
RDBMS client library
Python application (Embedded SQL)
Python DB adapter
RDBMS client library
Python application (little or no SQL)
Python ORM
Python DB adapter
RDBMS client library
Relational database (RDBMS)

Figure: Multitiered communication between application and database

- The figure illustrates the layers involved in writing a Python database application, with or without object relational mapping(ORM).
- The python applications can be integrated with some type of database system.
- The data can be retrieved and stored to / from RDBMS.
- A python DB adapter allows access database from python.
- It is a python module that enables users to interface with relational database’s client library.

Python ORM



Relational database (such as PostgreSQL or MySQL)

ID	FIRST_NAME	LAST_NAME	PHONE
1	John	Connor	+16105551234
2	Matt	Makai	+12025555689
3	Sarah	Smith	+19735554512
...	...	...	...

Python objects

```
class Person:
  first_name = "John"
  last_name = "Connor"
  phone_number = "+16105551234"
```

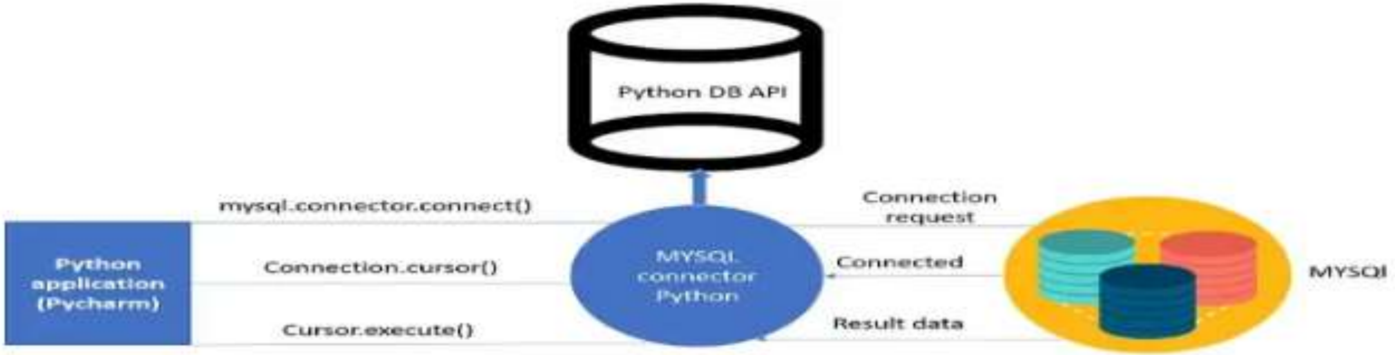
```
class Person:
  first_name = "Matt"
  last_name = "Makai"
  phone_number = "+12025555689"
```

```
class Person:
  first_name = "Sarah"
  last_name = "Smith"
  phone_number = "+19735554512"
```

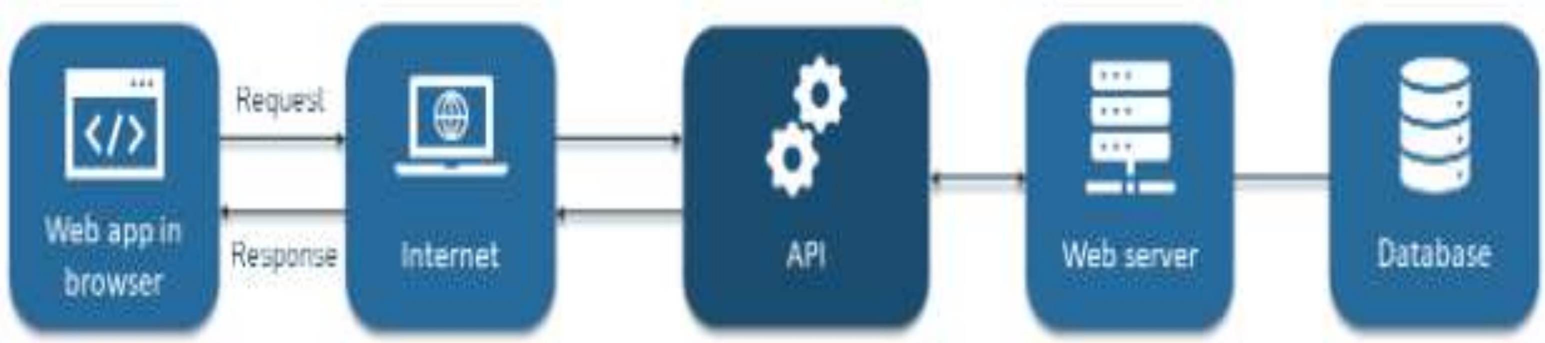
ORMs provide a bridge between **relational database tables, relationships and fields** and **Python objects**

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

- Database Application Programming Interface(DB-API) is Python's standard API used for accessing databases.
- The API is a specification which defines a set of objects and database access mechanisms to provide consistent access across the various database adapters and underlying database systems.



- The API is developed by SIG(Special Interest Group).
- The API provides the consistent interface to a variety of relational databases and porting code between different databases is simple and requires very less code.



2.1 Module Attributes

A DB-API compliant module must define the following global attributes.

Attributes	Description
apilevel	It indicates the version of API with which the module is compliant
threadsafety	It indicates the level of thread safety
paramstyle	SQL statement parameter style of this module
connect()	It is a function that is used to access the database through connection object
exception	The exceptions included in the module as globals are error, warning, database error etc.

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

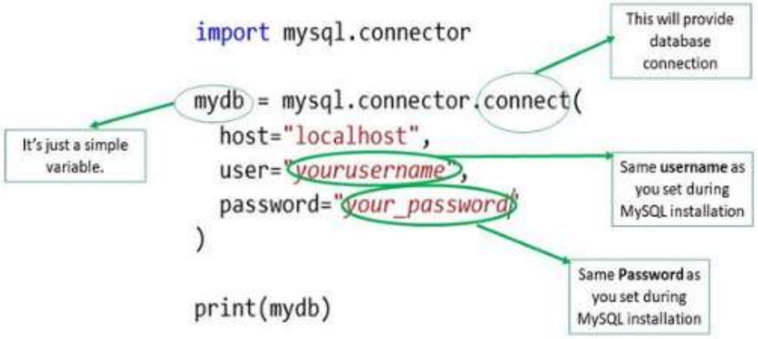
3: Fully threadsafe: threads can share the module, connections, and cursors

paramstyle database parameter styles

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

connect() function attributes:

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



```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="your_password"
)

print(mydb)
```

This will provide database connection

It's just a simple variable.

Same **username** as you set during MySQL installation

Same **Password** as you set during MySQL installation

DB-API Exception classes

Exception	Description
Error	Root error exception class
Warning	Root warning exception class
InterfaceError	Database interface error
DatabaseError	Database Error
OperationalError	Error during database operation execution
ProgrammingError	SQL command failed

1. Error

This is the base class for errors and a subclass to StandardError.

2. Warning

This is a subclass of StandardError. Python uses this for non-fatal issues.

3.InterfaceError

This is a subclass to Error and Python uses it for errors relating to the module for database access.

4.DatabaseError

This is a subclass to Error and Python uses it for database errors.

5. OperationalError

This is a subclass of DatabaseError. When Python loses connection to a database, it throws this error. This may happen when we haven’t selected a database.

6. ProgrammingError

This is a subclass of DatabaseError. Errors like bad table names cause this. This may happen when we try to create a duplicate database.

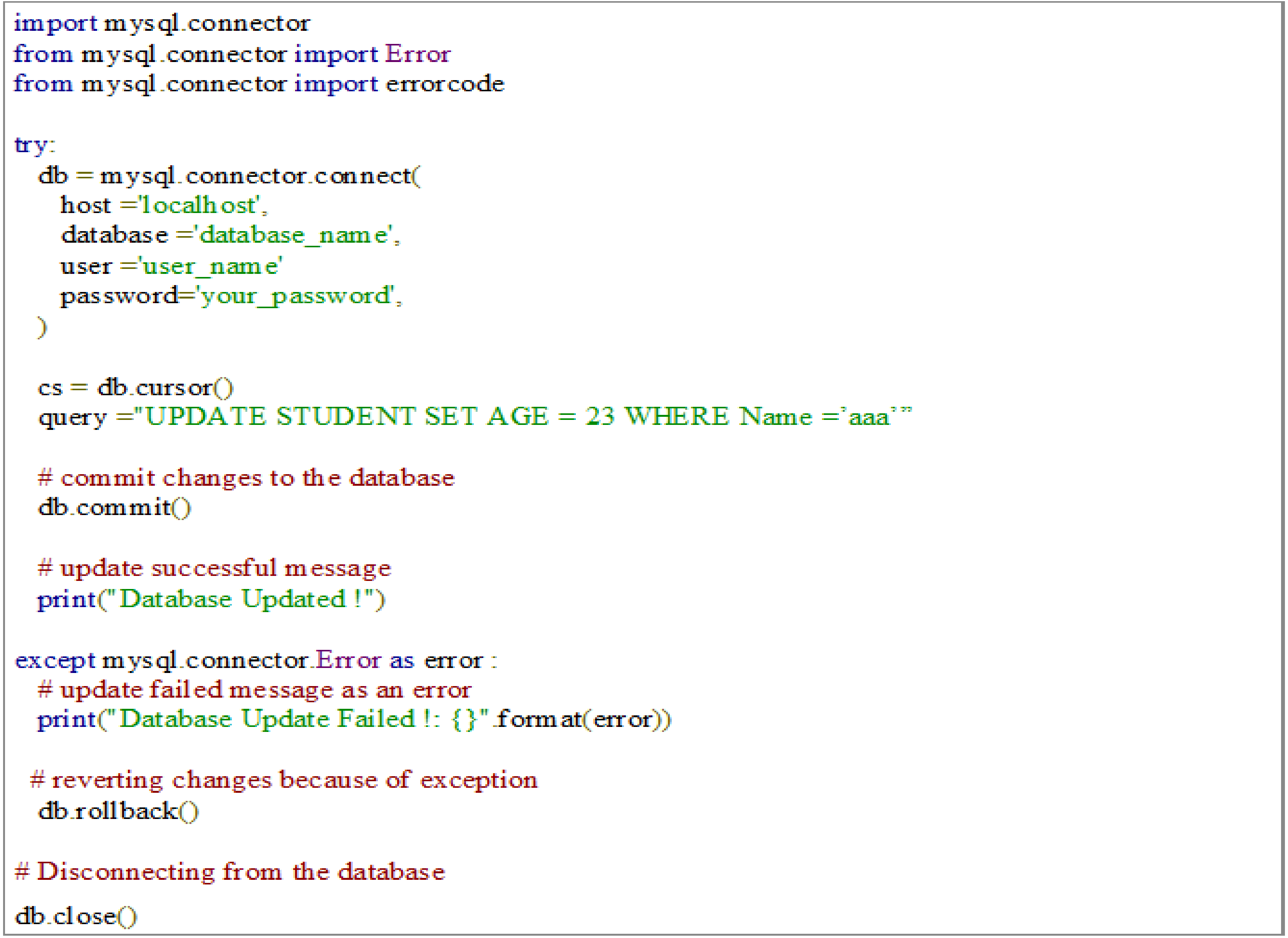
2.2 connection Objects

- Connections represents how the application communicates with the database.
- It represents the basic communication mechanism by which commands are sent to the server and the results returned.
- Once the connection is established, create cursors to send requests to and receive replies from the database.

connection Object Methods

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.

connection object example:



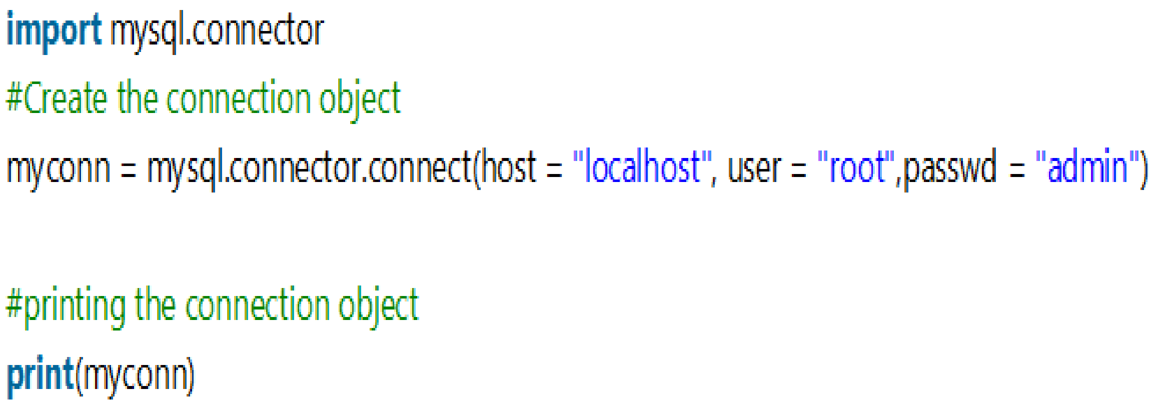
```
import mysql.connector
from mysql.connector import Error
from mysql.connector import errorcode

try:
  db = mysql.connector.connect(
    host ='localhost',
    database ='database_name',
    user ='user_name'
    password='your_password',
  )

  cs = db.cursor()
  query ="UPDATE STUDENT SET AGE = 23 WHERE Name ='aaa'"

  # commit changes to the database
  db.commit()

  # update successful message
  print("Database Updated !")

except mysql.connector.Error as error :
  # update failed message as an error
  print("Database Update Failed !: {}".format(error))

  # reverting changes because of exception
  db.rollback()

# Disconnecting from the database

db.close()
```

2.3 cursor Objects

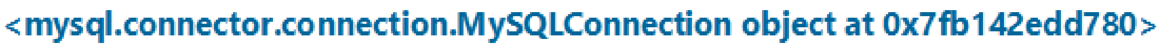
A cursor allows a user issue database command and retrieve rows resulting from queries.

Attribute	Description
arraysize	Number of row to fetch at a time with fetchmany(); defaults to 1
connection	Connection that created this cursor
close()	Closes cursor
fetchone()	Fetches the next row of query result
fetchall()	Fetches all rows of a query result.
messages	List of messages received from the database for cursor execution.
rowcount	Number of rows that the last execute produced or affe ted
execute(op[,args])	Execute a database query or command

- Connection object

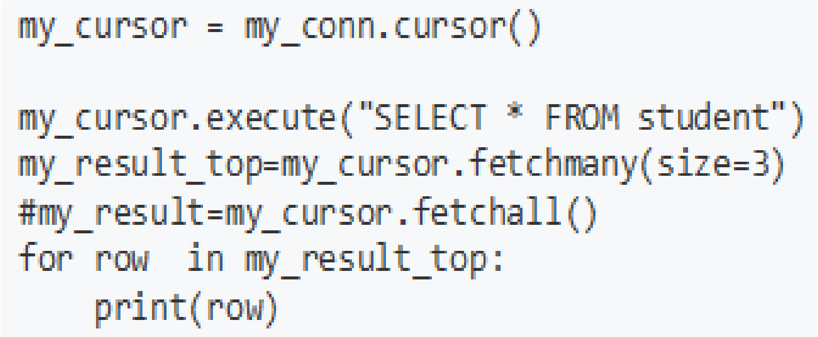


```
import mysql.connector
#Create the connection object
myconn = mysql.connector.connect(host = "localhost", user = "root",passwd = "admin")

#printing the connection object
print(myconn)
```

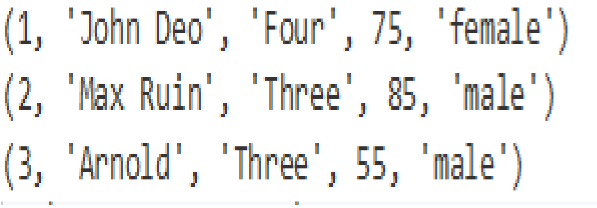


```
<mysql.connector.connection.MySQLConnection object at 0x7fb142edd780>
```

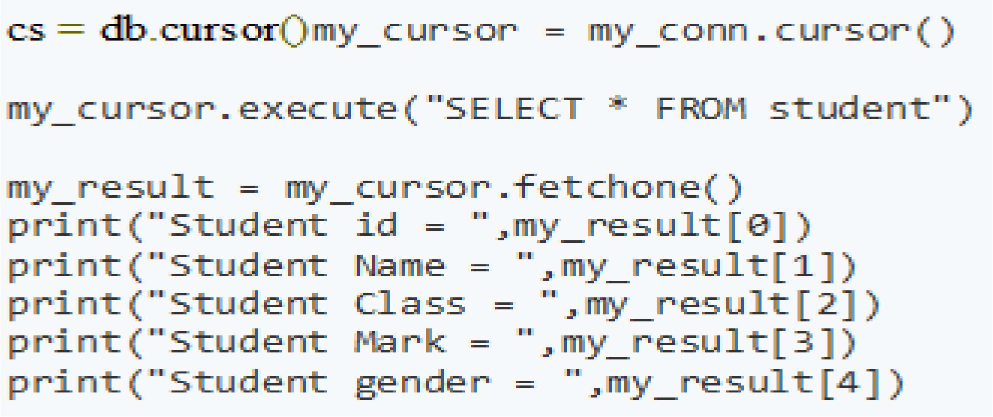
Cursor object example:



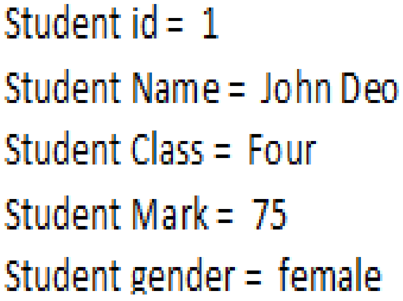
```
my_cursor = my_conn.cursor()

my_cursor.execute("SELECT * FROM student")
my_result_top=my_cursor.fetchmany(size=3)
#my_result=my_cursor.fetchall()
for row  in my_result_top:
    print(row)
```



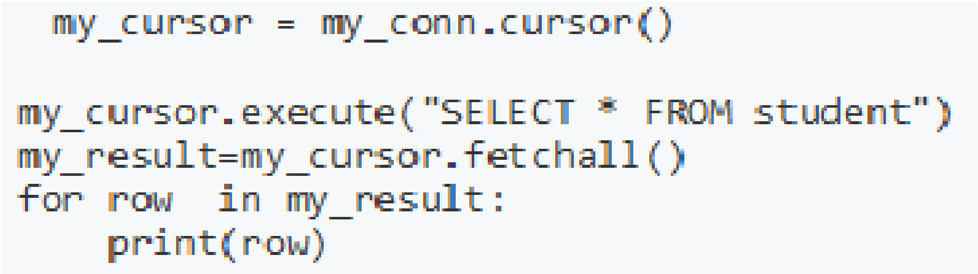
```
(1, 'John Deo', 'Four', 75, 'female')
(2, 'Max Ruin', 'Three', 85, 'male')
(3, 'Arnold', 'Three', 55, 'male')
```



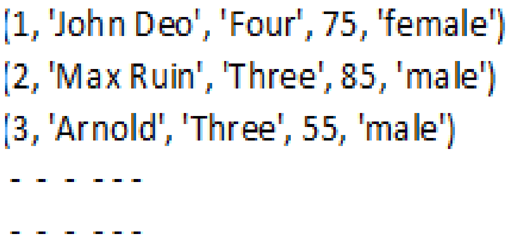
```
cs = db.cursor()my_cursor = my_conn.cursor()

my_cursor.execute("SELECT * FROM student")

my_result = my_cursor.fetchone()
print("Student id = ",my_result[0])
print("Student Name = ",my_result[1])
print("Student Class = ",my_result[2])
print("Student Mark = ",my_result[3])
print("Student gender = ",my_result[4])
```



```
Student id =  1
Student Name =  John Deo
Student Class =  Four
Student Mark =  75
Student gender =  female
```



```
  my_cursor = my_conn.cursor()

my_cursor.execute("SELECT * FROM student")
my_result=my_cursor.fetchall()
for row  in my_result:
    print(row)
```



```
(1, 'John Deo', 'Four', 75, 'female')
(2, 'Max Ruin', 'Three', 85, 'male')
(3, 'Arnold', 'Three', 55, 'male')
- - - - ---
- - - - ---
```

2.4 Type Objects and Constructors:

- The parameters 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.
- The DB-API needs to create constructors that build special objects that can easily be converted to the appropriate database objects.

Type object and constructors

Type object	Description
Date(yr, mo, dy)	Object for a date value
Time(hr, min, sec)	Object for a time value
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

2.5 Relational Databases

The database systems that are accessible to interfaces in python are as follows:

Commercial RDBMS	Open Source RDBMS	Database APIs
•Informix •Sybase •Oracle •MS SQL Server •DB/2 •SAP •Interbase •Ingres	•MySQL • PostgreSQL • SQLite • Gadfly	•JDBC • ODBC

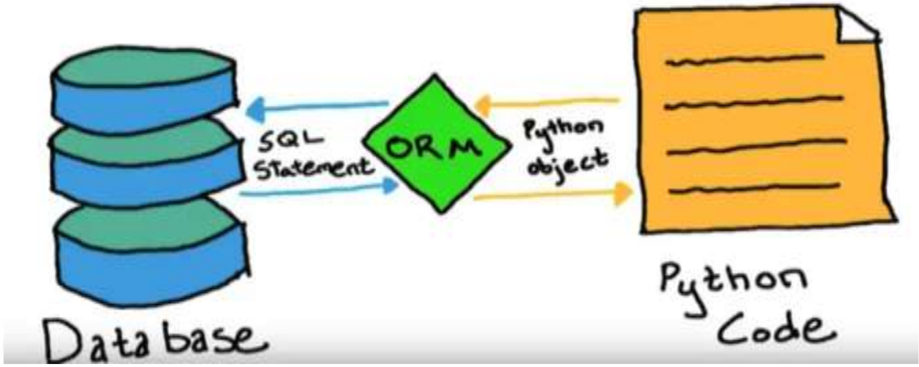
2.6 Databases and Python: Adapters

- For each database there exists one or more adapters
- The adapter is responsible for the connection between the target database system and python.
- The databases like SQLServer, Sybase, SAP, Oracle have more than one adapter.
- If there are multiple adapters available for any database system then the best among them must be selected based on their features.

3.Object Relational Managers(ORMs)

- There are variety of databases systems available and most of them have python interfaces to utilize their power.
- But the issue is that those systems need minimum knowledge of SQL.
- If the programmer can manipulate python objects instead of SQL queries and wants to use database as backend, then ORM

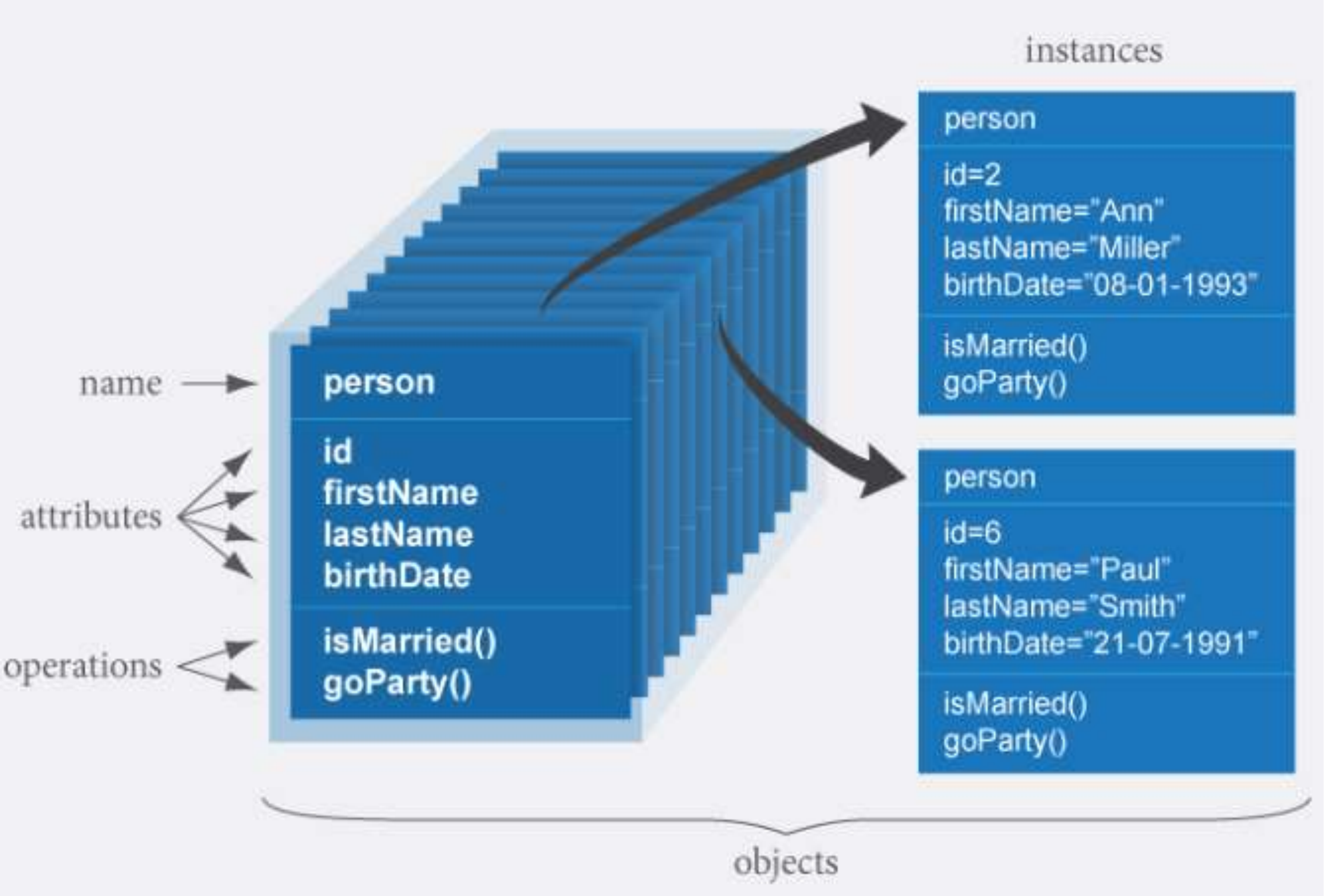
  is the best choice.



3.1 Think objects, Not SQL

- Using ORMs, database tables are converted into Python classes with columns and features as attributes and methods as database operations.
- The application can be set to an ORM.
- Certain operations might be complex and require more lines of code than using adapter directly, because ORM’s perform a lot of work on behalf of users.

Data representation in object oriented programming



2.2 Python and ORMs:

The popular ORMs today are

- SQLAlchemy
- SQLObject

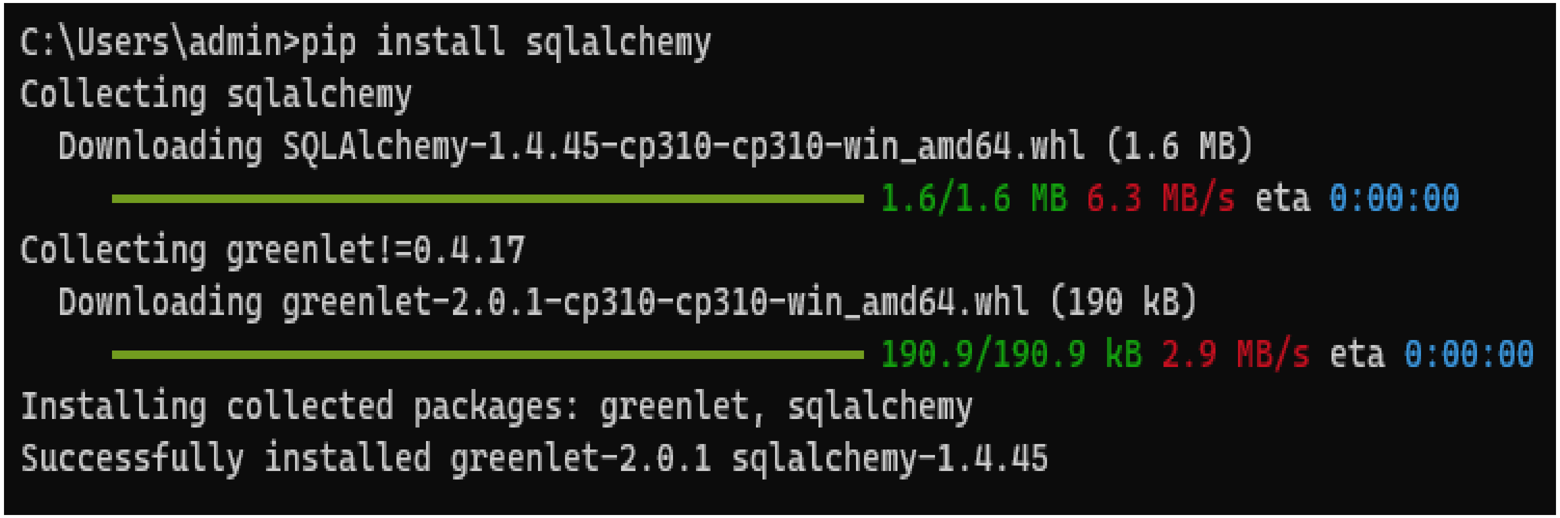
Some other ORMs include

- PyDO/PyDO2
- PDO
- Dejavu
- Durus
- Qlime
- ForgetSQL

SQLAlchemy ORM:

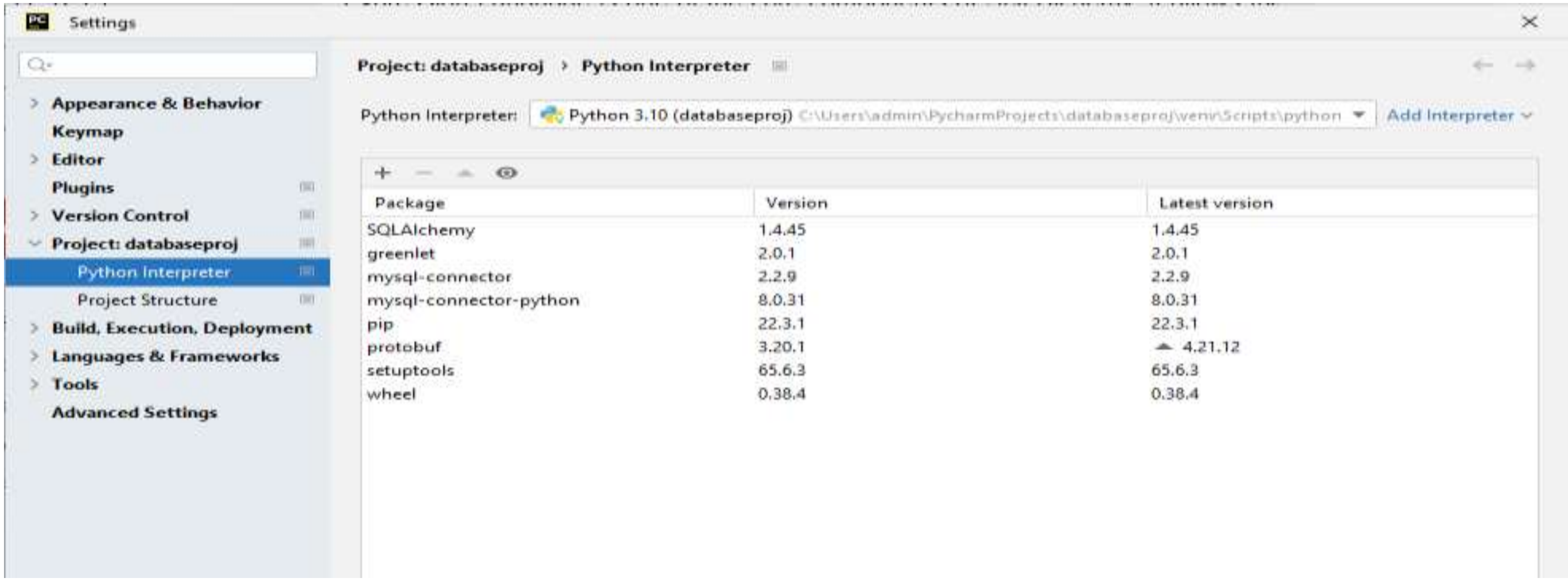
- It is the Python SQL toolkit and Object Relational Mapper.
- It is written in Python.
- It maps classes to the tables in databases.
- It is a library which provides the communication between the Python programs and databases.
- SQLAlchemy includes dialects for SQLite, MySQL, Oracle, Firebird, Sybase and others.

SQLAlchemy installation



```
C:\Users\admin>pip install sqlalchemy
Collecting sqlalchemy
  Downloading SQLAlchemy-1.4.45-cp310-cp310-win_amd64.whl (1.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 6.3 MB/s eta 0:00:00
Collecting greenlet!=0.4.17
  Downloading greenlet-2.0.1-cp310-cp310-win_amd64.whl (190 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 190.9/190.9 kB 2.9 MB/s eta 0:00:00
Installing collected packages: greenlet, sqlalchemy
Successfully installed greenlet-2.0.1 sqlalchemy-1.4.45
```

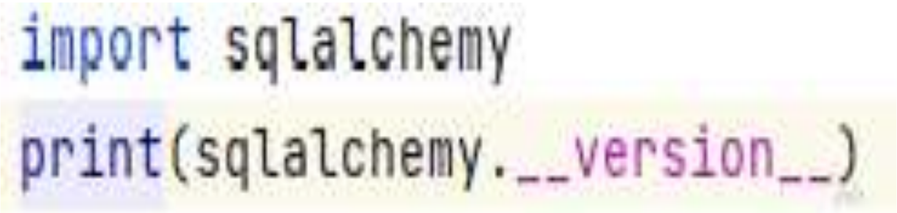
SQLAlchemy package/module added to pycharm



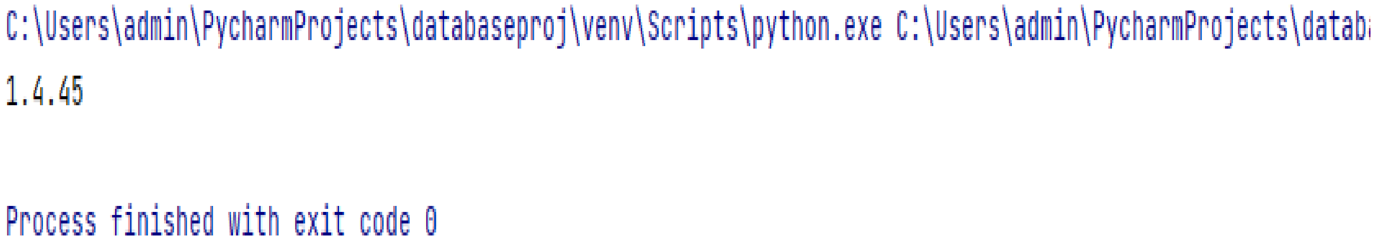
Python Interpreter: Python 3.10 (databaseproj) C:\Users\admin\PycharmProjects\databaseproj\venv\Scripts\python

Package	Version	Latest version
SQLAlchemy	1.4.45	1.4.45
greenlet	2.0.1	2.0.1
mysql-connector	2.2.9	2.2.9
mysql-connector-python	8.0.31	8.0.31
pip	22.3.1	22.3.1
protobuf	3.20.1	▲ 4.21.12
setuptools	65.6.3	65.6.3
wheel	0.38.4	0.38.4

To check the version of SQLAlchemy:



```
import sqlalchemy
print(sqlalchemy.__version__)
```

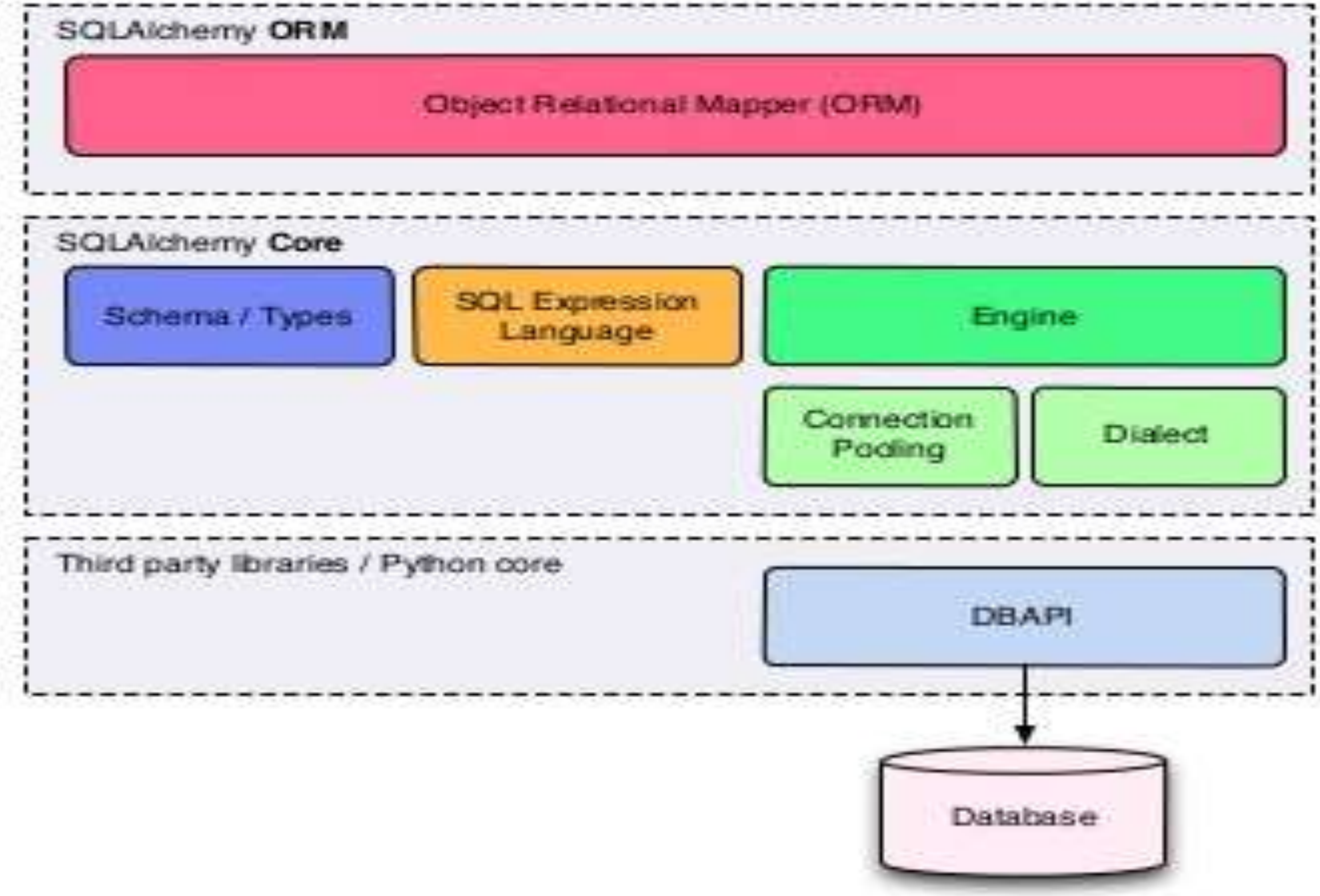


```
C:\Users\admin\PycharmProjects\databaseproj\venv\Scripts\python.exe C:\Users\admin\PycharmProjects\datab
1.4.45


Process finished with exit code 0
```

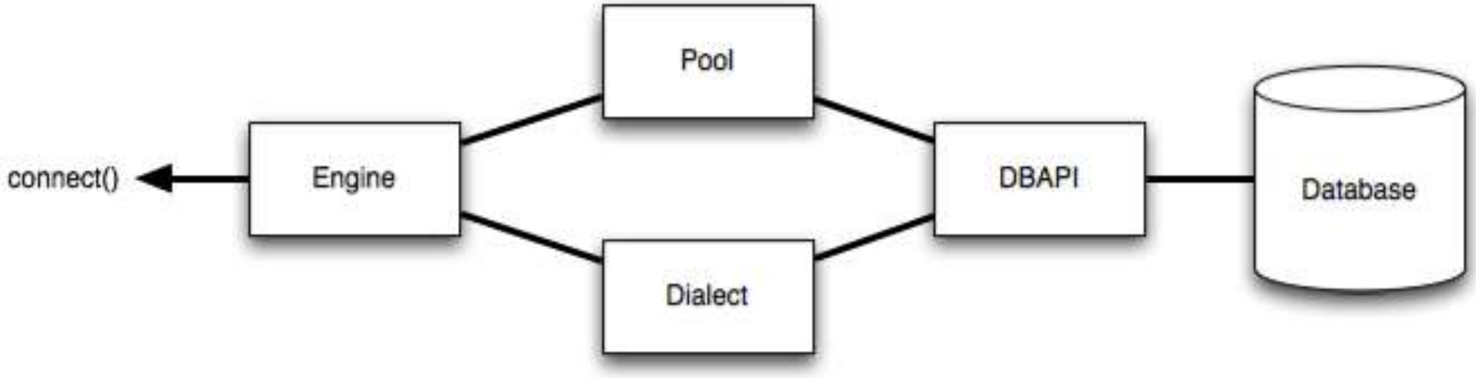
To connect to a database:

- A connection pool consists of long running connections in memory for efficient re-use, managing the total number of connections an application might use simultaneously.

- The SQL dialect, obtained from the Structured Query Language, uses human-readable expressions to define query statements.

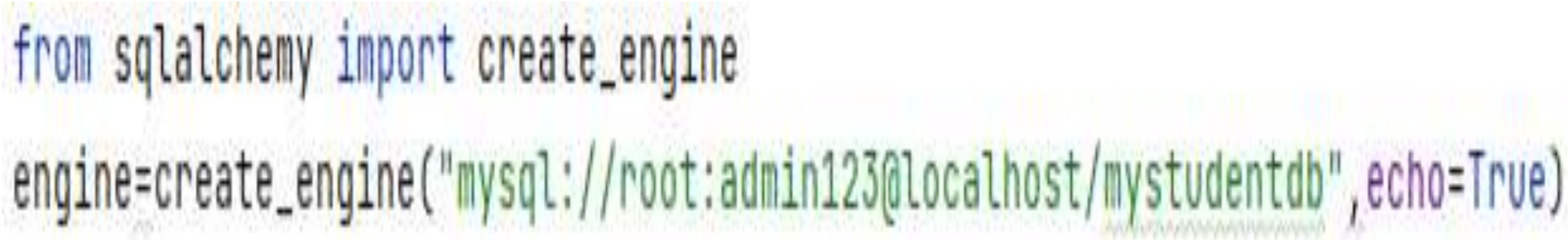


- Engine class: it connects a Pool and Dialect together



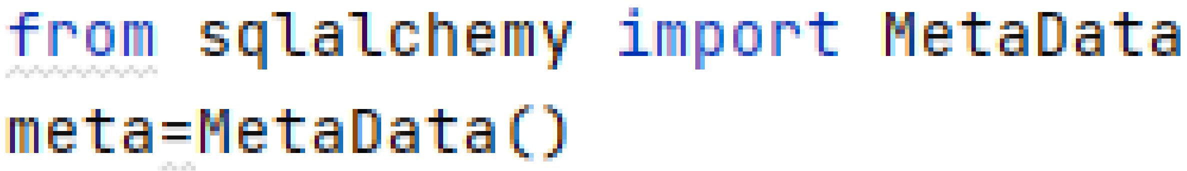
- The Engine is the starting point for any SQLAlchemy application.
- It's “home base” for the actual database and its DBAPI, delivered to the SQLAlchemy application through a connection pool and a Dialect, which describes how to talk to a specific kind of database/DBAPI combination.

- The create_engine function returns an instance of an engine; however, it does not actually open a connection until an action is called that would require a connection, such as a query.



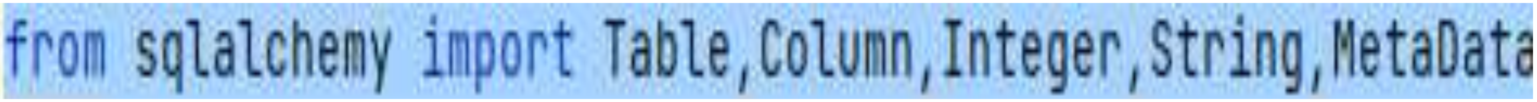
```
from sqlalchemy import create_engine

engine=create_engine("mysql://root:admin123@localhost/mystudentdb",echo=True)
```

- Metadata contains definitions of tables and associated objects.



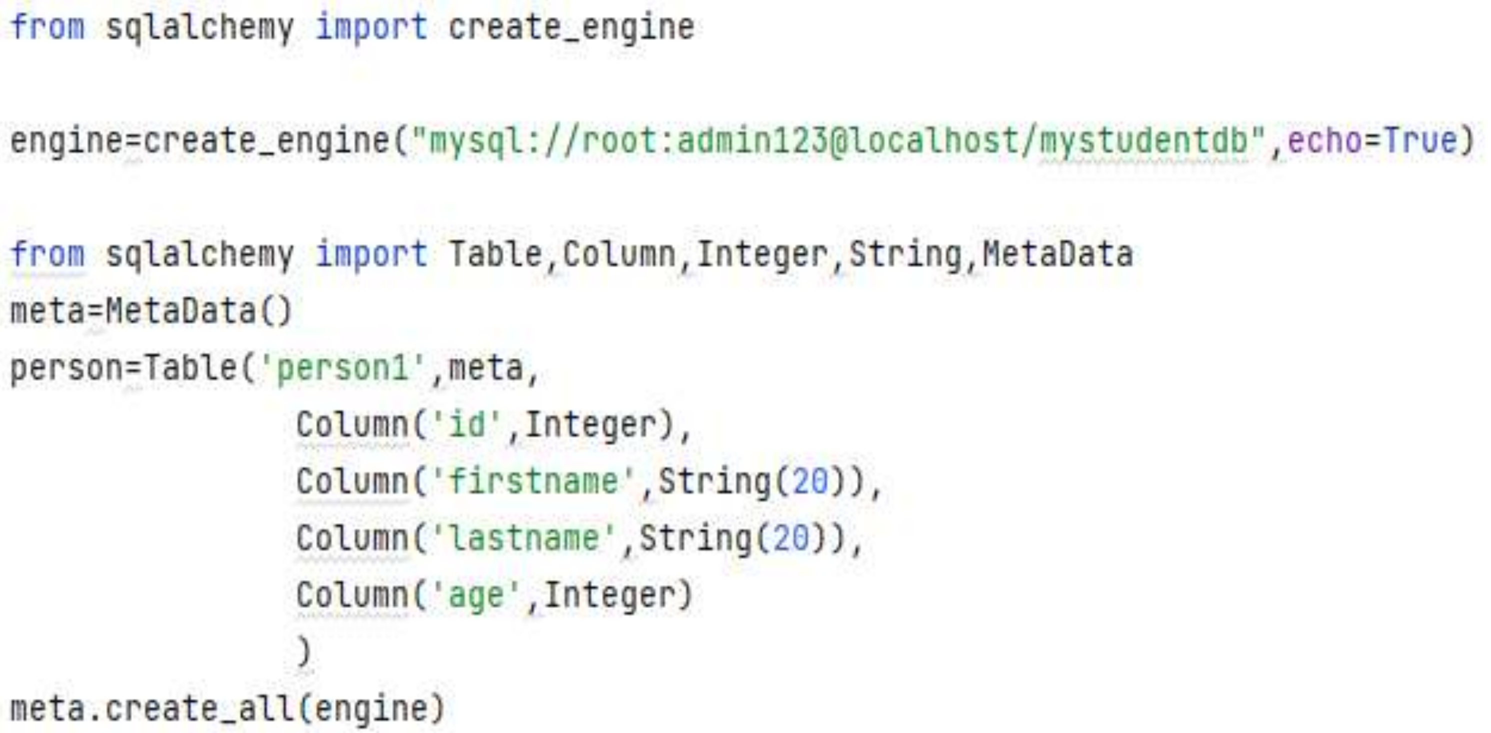
```
from sqlalchemy import MetaData
meta=MetaData()
```

- SQLAlchemy Column object represents a column in a database table which is in turn represented by a Tableobject.



```
from sqlalchemy import Table,Column,Integer,String,MetaData
```

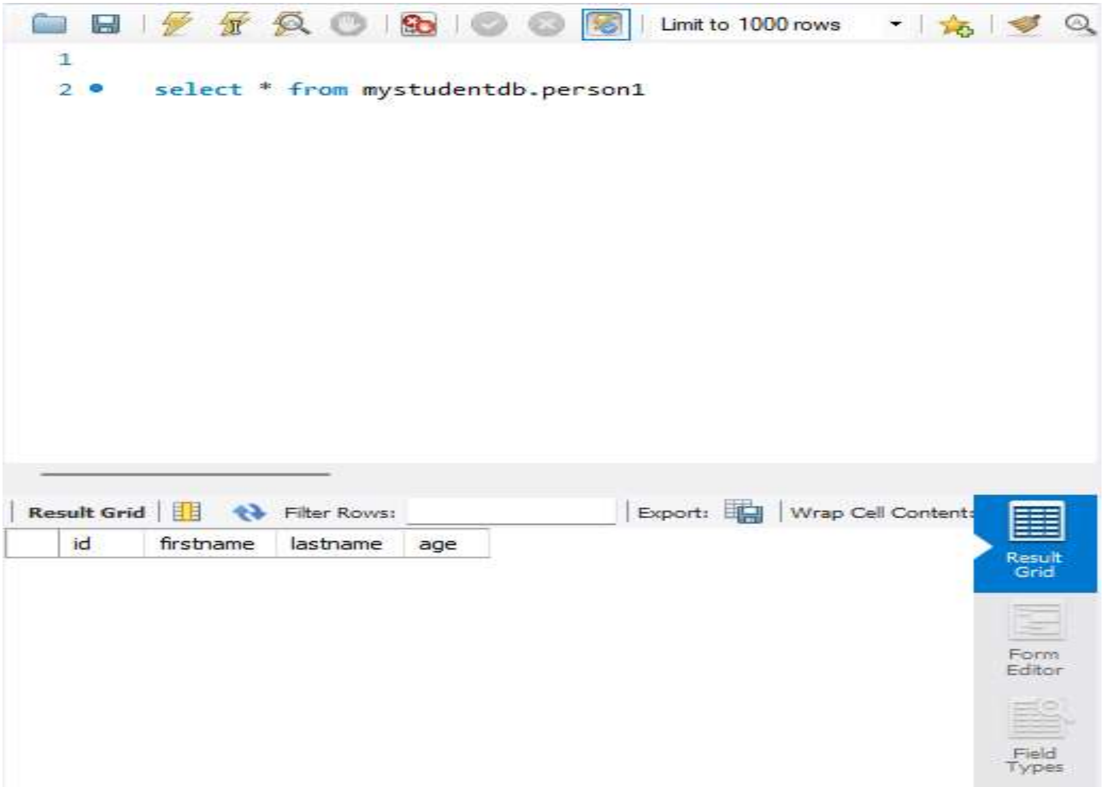
To create table named person1 in mystudentdb:



```
from sqlalchemy import create_engine

engine=create_engine("mysql://root:admin123@localhost/mystudentdb",echo=True)

from sqlalchemy import Table,Column,Integer,String,MetaData
meta=MetaData()
person=Table('person1',meta,
             Column('id',Integer),
             Column('firstname',String(20)),
             Column('lastname',String(20)),
             Column('age',Integer)
             )
meta.create_all(engine)
```

- If ‘echo = True’

```
CREATE TABLE person1 (
    id INTEGER,
    firstname VARCHAR(20),
    lastname VARCHAR(20),
    age INTEGER
)
```

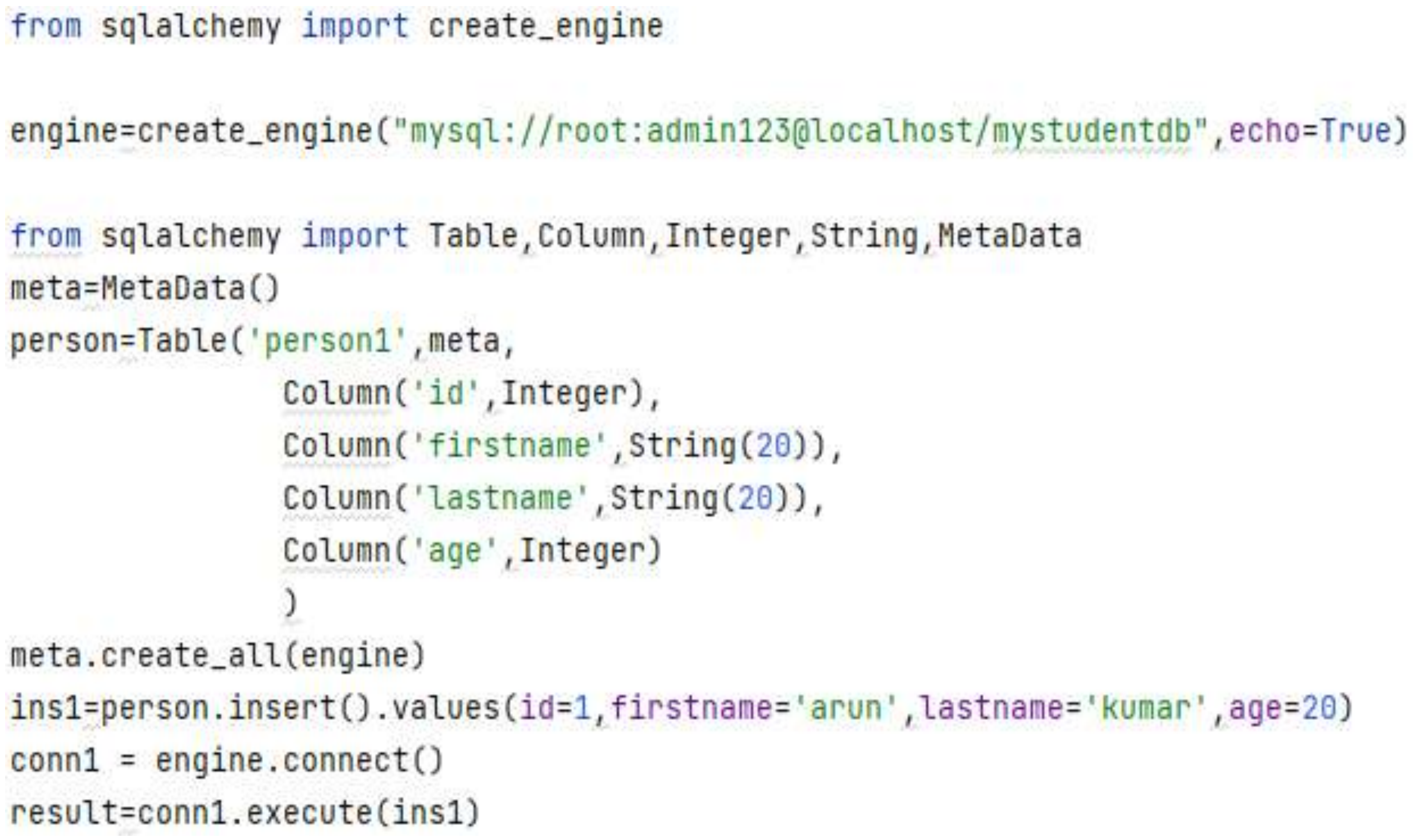


```
select * from mystudentdb.person1
```

	id	firstname	lastname	age

To insert into table:

The INSERT statement is created by executing insert() method as follows –



```
from sqlalchemy import create_engine

engine=create_engine("mysql://root:admin123@localhost/mystudentdb",echo=True)

from sqlalchemy import Table,Column,Integer,String,MetaData
meta=MetaData()
person=Table('person1',meta,
               Column('id',Integer),
               Column('firstname',String(20)),
               Column('lastname',String(20)),
               Column('age',Integer)
               )
meta.create_all(engine)
ins1=person.insert().values(id=1,firstname='arun',lastname='kumar',age=20)
conn1 = engine.connect()
result=conn1.execute(ins1)
```

4. Related Modules

- The most commonly used related modules and databases in database programming are as follows:

Databases		
MYSQL	PostgreSQL	SQLObject
Gadfly	Sapdb	sybase
SQLite	Pymssql	oracle
SQLserver	PoPy	SQLAlchemy
Fire bird(interbase)	pymssql	PyDO/PyDO2 SQL object