UNIT-5

UNIT – V

Database Programming: Introduction, Python Database Application Programmer’s Interface (DB-API), Object Relational Managers (ORMs), Related Modules

Database Programming: Introduction

✓ The Python programming language has powerful features for **database programming**

✓ Python supports various **databases like MySQL, Oracle, Sybase, PostgreSQL**, etc.

✓ Python also supports Data Definition Language (DDL), Data Manipulation Language (DML) and Data Query Statements.

✓ For database programming, the **Python DB API** is a widely used module that provides a database application programming interface.

✓The Python standard for **database interfaces** is the **Python DB-API**.

✓You can choose the right database for your application.

✓Python Database API supports a wide range **of database servers** such as −

GadFly

mSQL

MySQL

PostgreSQL

Microsoft SQL Server 2000

Informix

Interbase

Oracle

Sybase

Benefits of Python for database programming

✓Programming in Python is arguably **more efficient and faster compared** to other languages.

✓Python is famous for its **portability.**

✓It is **platform independent.**

✓Python supports **SQL cursors**.

✓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 s**upports relational database systems.

✓Python **database APIs** are compatible with **various databases,** so it is very easy to migrate and port **database application interfaces.**

✓We must **download** a separate **DB API module** for each database we need to access.

For example, if you need to access an Oracle database as well as a MySQL database, you must download both the Oracle and the MySQL database modules.

✓The DB API provides a **minimal standard for working with databases** using Python structures and syntax wherever possible.

This API includes the following –

1. Importing the API module.
2. Acquiring a connection with the database.
3. Issuing SQL statements and stored procedures.
4. Closing the connection

Python Database.

✓Data is retrieved from a database system using the **SQL language**.

✓Data is **everywhere and software applications** use that. Data is either in memory, files or databases.

✓Python has **bindings** for many database systems including **MySQL, Postregsql, Oracle, Microsoft SQL Server and Maria DB.**

✓One of these database management systems (DBMS) is called **SQLite.**

✓**SQLite** was created in the year **2000** and is one of the many management systems in the database zoo.

SQL:

SQL is a special-purpose programming language designed for managing data held in a databases.

- ✓**Database commands and queries** are given to a database by SQL.
- ✓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.

Here are some examples of SQL commands.

1. Creating a Database

CREATE DATABASE test;

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

- ✓The first line **creates a database** named **"test,"** and
- ✓Assuming that you are a database administrator, the **second line** can be used to **grant permissions** to specific users (or all of them) so that they can perform the database operations

2. Using a Database

```
USE test;
```

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

3.Dropping a Database

```
DROP DATABASE test;
```

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

4.Creating a Table

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

This statement creates a new table with a string column **login** and a pair of integer fields **uid** and **prid**.

5. Dropping a Table

```
DROP TABLE users;
```

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

6. Inserting a Row

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

You can insert a new row in a database with the **INSERT** statement. Specify the table and the values that go into each field. For our example, the string 'leanna' goes into the login field, and 311 and 1 to uid and prid, respectively.

7. Updating a Row

```
UPDATE users SET prid=4 WHERE prid=2;
UPDATE users SET prid=1 WHERE uid=311;
```

To change existing table rows, you use the UPDATE statement. Use **SET** for the columns that are changing and provide any criteria for determining **which rows should change**.

In the first example, all users with a "**project ID"** or prid of 2 will be moved to project #4. In the second example, we take one user (with a UID of 311) and move them to project #1.

8. Deleting a Row

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

To delete a table row, use the DELETE FROM command, give the table you want to delete rows from, and any optional criteria. Without it, as in the second example, all rows will be deleted.

Databases and Python:

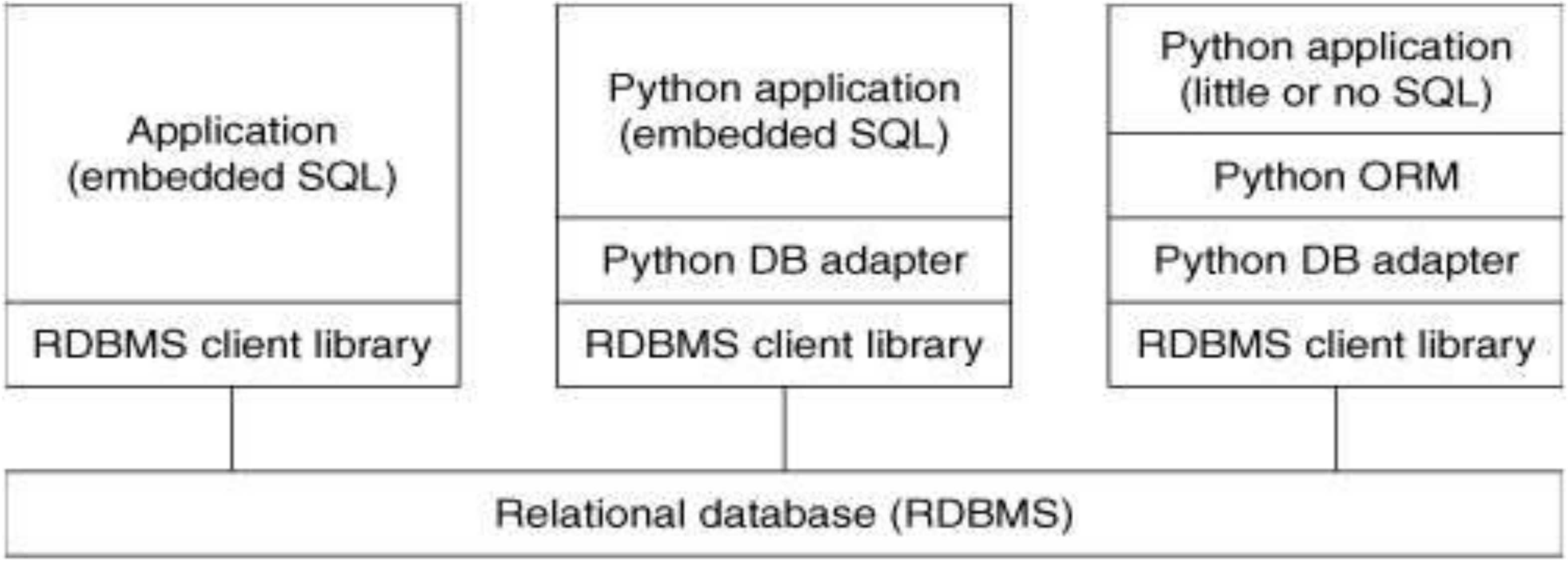
✓We can access relational databases from Python, either directly through **a database interface**, or via an **ORM**.

✓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 adapt to the Python DB-SIG's Application Programmer Interface (API).

✓**ORM** is a code library that automates the transfer of data stored in relational databases tables into objects that are more commonly used in application code.


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)

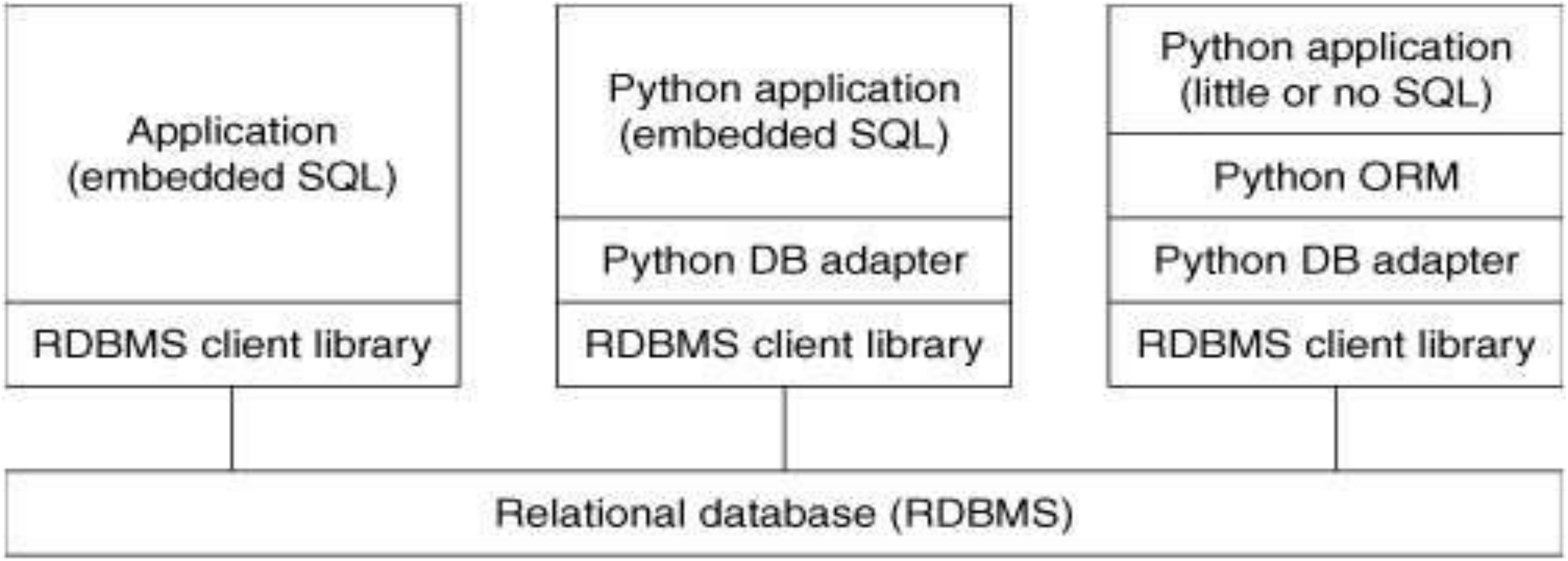
✓Figure represents **“Multitiered communication between application and database.”**

✓The figure illustrates **the layers involved in writing a Python database application, with and without an ORM.**

✓ the **DB-API** is your interface to the C libraries of the database client.

✓The first box is generally a C/C++ program while **DB-API compliant adapters allow you program applications in Python.**

✓ORMs can simplify an **application by handling all of the database-specific details**.



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

What is the 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 DB-API is a specification for a **common interface to relational databases.**

✓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.**

✓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, usually only requiring tweaking several lines of code. The current version of the specification is **version 2.0.**

Module Attributes

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

DB-API Module Attributes:

I. Data Attributes:

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

1. Apilevel:

✓ String constant stating the **supported DB API level.**

✓ Currently only the strings **"1.0"** and **"2.0"** are allowed.

✓ If not given, a **DB-API 1.0 level interface should be assumed**.

2. 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**.

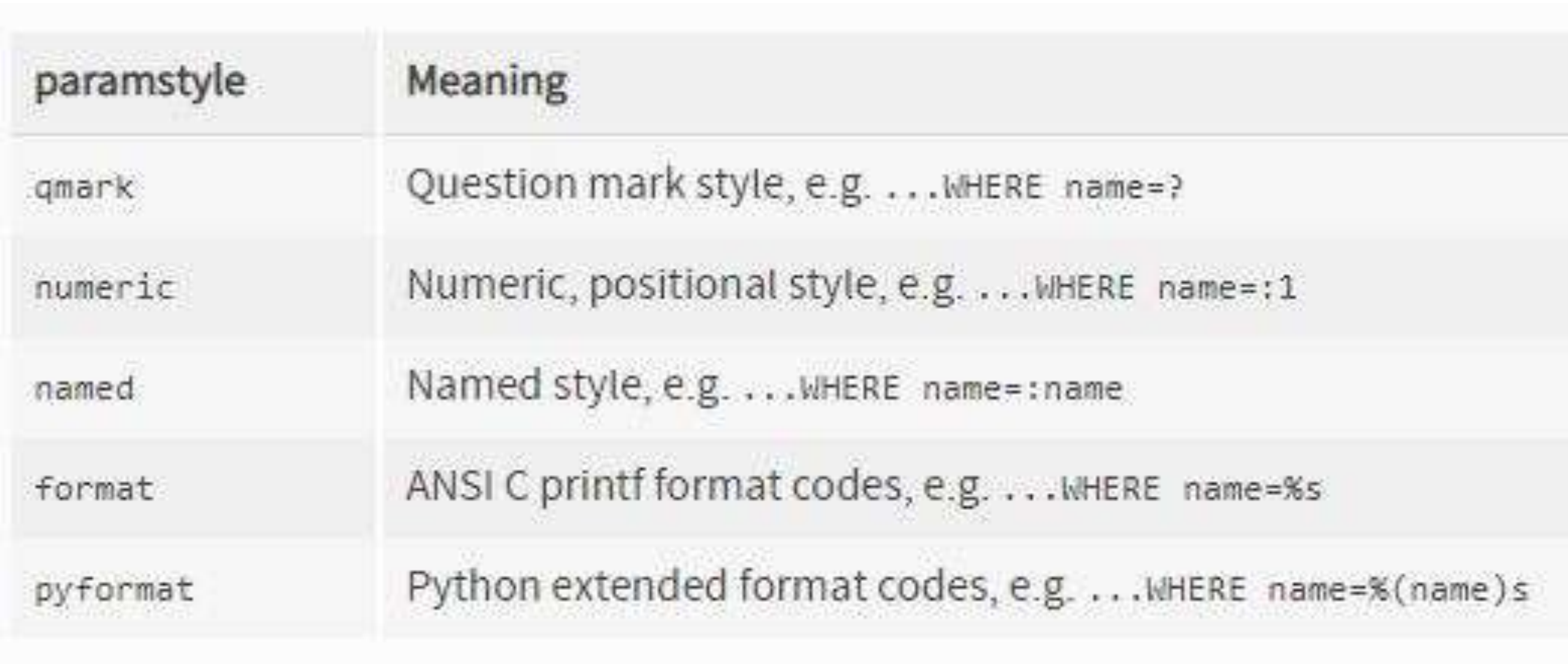
2: Moderately threadsafe: threads can share the **module** and **connections** **but not cursors.**

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

3.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 ,**we will use when **building rows for a query or command**.



paramstyle	Meaning
`qmark`	Question mark style, e.g. `...WHERE name=?`
`numeric`	Numeric, positional style, e.g. `...WHERE name=:1`
`named`	Named style, e.g. `...WHERE name=:name`
`format`	ANSI C printf format codes, e.g. `...WHERE name=%s`
`pyformat`	Python extended format codes, e.g. `...WHERE name=%(name)s`

II. Function Attribute(s):

1. connect()

✓ **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.**

connect() Function Attributes:

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

✓We can pass in database connection information **as a string** with multiple parameters (DSN) or individual parameters passed as positional arguments or more likely, keyworded arguments.

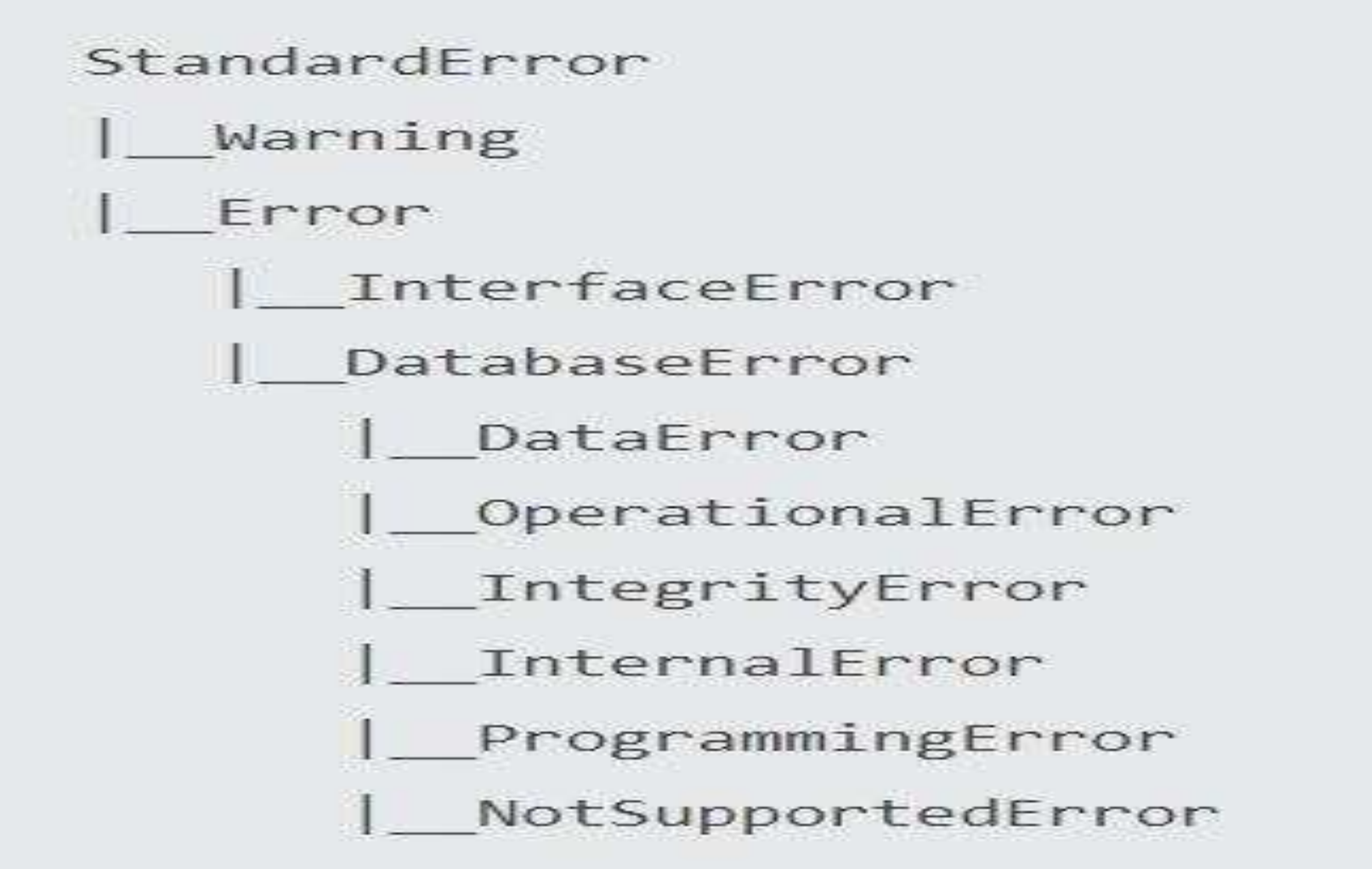
✓ Here is an example of **using connect():**

```
connect(dsn='myhost:MYDB',user='guido',password='234$')
```

III. Exceptions

The module should make **all error information** available through these exceptions or subclasses.

DB-API Exception Classes hierarchy



```
StandardError
 |__Warning
 |__Error
      |__InterfaceError
      |__DatabaseError
           |__DataError
           |__OperationalError
           |__IntegrityError
           |__InternalError
           |__ProgrammingError
           |__NotSupportedError
```

1. **Warning:**

   - ✓ Exception raised for important warnings **like data truncations** while inserting, etc.

2. **Error:**

   - ✓ Exception that is the **base class of all other error exceptions**. We can use this to **catch all errors** with one single except statement

3. **InterfaceError:**

   - ✓ Exception raised for errors that are **related to the database** interface rather than the database itself.

4. **DatabaseError:**

   - ✓ Exception raised for **errors that are related to the database**

5. **DataError:**

   ✓ Exception raised for errors that are **due to problems with the processed data** like division by zero, numeric value out of range, etc.

6. OperationalError: Error during database operation execution

7. IntegrityError: Database relational integrity error

8. InternalError: Error that occurs within the database

9. ProgrammingError: SQL command failed

10. NotSupportedError: Unsupported operation occurred

IV. Connection Objects

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

V. Cursor Objects:

✓Once you have a connection, you can **start talking to the database.**

✓ Cursor allows a **user issue database commands and** retrieve rows resulting from Queries.

✓ 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.**

✓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.**

In python, cursor objects having data attributes and methods.

Cursor Object Attributes:

Object Attribute	***Description***
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);**
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

callproc(***func[, args])***	*Call a stored procedure*
arraysize	Number of rows to fetch at a time with fetch many(); defaults to 1
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 arguments
messages	List of messages (set of tuples) received from the database for cursor execution (optional)

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
setinput-sizes(sizes)	Set maximum input-size allowed

VI. Type Objects and Constructors

✓There is a fine line between **Python objects and native database objects.**

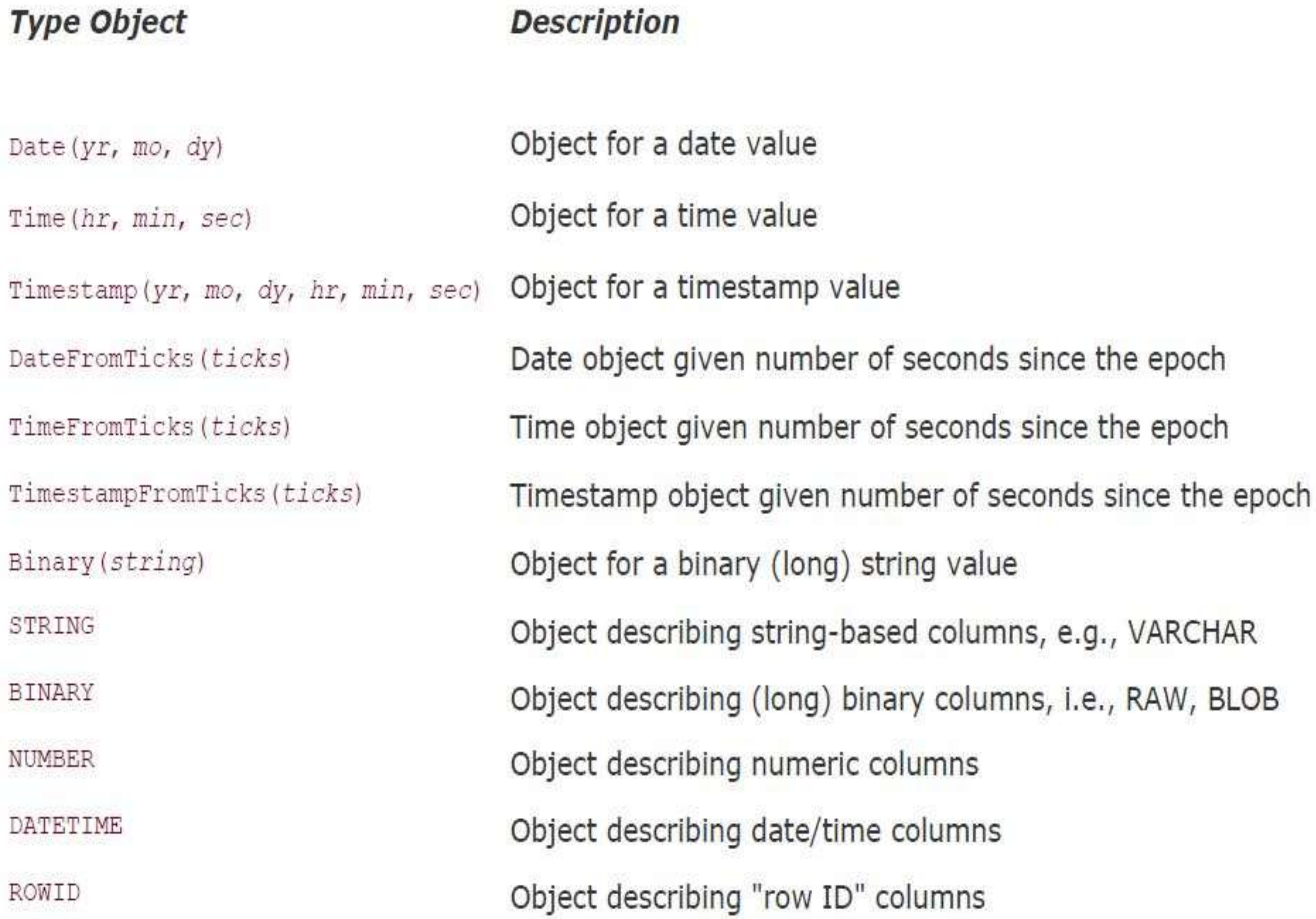
✓As a programmer writing to Python's DB-API, the parameters you send to a database are given as strings, but the **database may need to convert it to a variety of different, supported data types** that are correct for any particular query.

✓For example, should the Python string be converted to a VARCHAR, a TEXT, a BLOB, or a raw BINARY object.

? Care must be taken to provide database input in the expected.

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

Type Objects and Constructors:



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

Relational Databases

Commercial RDBMSs

- Informix
- Sybase
- Oracle
- MS SQL Server
- DB/2
- SAP
- Interbase
- Ingres

Open Source RDBMSs

- MySQL
- PostgreSQL
- SQLite
- Gadfly

Database APIs

- JDBC
- ODBC

Changes to API Between Versions

Several important changes were made when the DB-API was revised from version 1.0 (1996) to 2.0 (1999):

- Required dbi module removed from API
- Type objects were updated
- New attributes added to provide better database bindings
- callproc() semantics and return value of execute() redefined
- Conversion to class-based exceptions

Next version of the DB-API, tentatively named DB-API 3.0. These include the following:

- Better return value for nextset() when there is a new result set
- Switch from float to Decimal
- Improved flexibility and support for parameter styles
- Prepared statements or statement caching
- Refine the transaction model
- State the role of API with respect to portability
- Add unit testing

Databases and Python: Adapters

✓For each of the databases supported, **there exists one or more adapters that allows you connect to the target database system from Python.**

✓Some databases, such as Sybase, SAP, Oracle, and SQLServer, have more than one adapter available.

✓The best thing to do is to find out which ones fit your needs best.

- **how good its performance is,**
- **how useful is its documentation and/or Web site,**
- **whether it has an active community or not,**
- **what the overall quality and stability of the driver is, etc.**

Ex:

For MySQL –only one Python adapter i.e., **MySQLdb**

For PostgreSQL—three Python adapter is available i.e., psycopg, PyPgSQL, and PyGreSQL

For SQLite --- only one Python adapter i.e.,**sqlite3**

Examples of Using Database Adapters:

MySQL:

- ✓only MySQL Python adapter: **MySQLdb**

We first log in as an administrator to create a database and grant permissions, then log back in as a normal client.

```
>>> import MySQLdb
>>> cxn = MySQLdb.connect(user='root')
>>> cxn.query('DROP DATABASE test')
        Traceback (most recent call last):
        File "<stdin>", line 1, in ?
        _mysql_exceptions.OperationalError: (1008, "Can't drop database 'test';
        database doesn't exist")
>>> cxn.query('CREATE DATABASE test')
>>> cxn.query("GRANT ALL ON test.* to ''@'localhost'")
>>> cxn.commit()
>>> cxn.close()
```

✓In the code above, **we did not use a cursor**. Some adapters have **Connection objects,** which can execute **SQL queries** with the **query() method**, but not all.

✓The **commit()** was optional for us as auto-commit is turned on by **default in MySQL**. We then **connect back to the new database as a regular user**, create a table, and perform the usual queries and commands using SQL.

creating a table:

- ✓ This time we use **cursors and their execute() method.**

```
>>> cxn = MySQLdb.connect(db='test')
>>> cur = cxn.cursor()
>>> cur.execute('CREATE TABLE users(login VARCHAR(8), uid INT)')
    0L
```

Now we will insert a few rows into the database and query them out.

```
>>> cur.execute("INSERT INTO users VALUES('john', 7000)")
1L
>>> cur.execute("INSERT INTO users VALUES('jane', 7001)")
1L
>>> cur.execute("INSERT INTO users VALUES('bob', 7200)")
1L
>>> cur.execute("SELECT * FROM users WHERE login LIKE 'j%'")
2L
>>> for data in cur.fetchall():
... print '%s\t%s' % data
...
john 7000
jane 7001
```

updating or deleting rows:

```
>>> cur.execute("UPDATE users SET uid=7100 WHERE uid=7001")

1L

>>> cur.execute("SELECT * FROM users")

3L

>>> for data in cur.fetchall():

... print '%s\t%s' % data

john 7000

jane 7100

bob 7200

>>> cur.execute('DELETE FROM users WHERE login="bob"')

1L

>>> cur.execute('DROP TABLE users')

0L

>>> cur.close()

>>> cxn.commit()

>>> cxn.close()
```

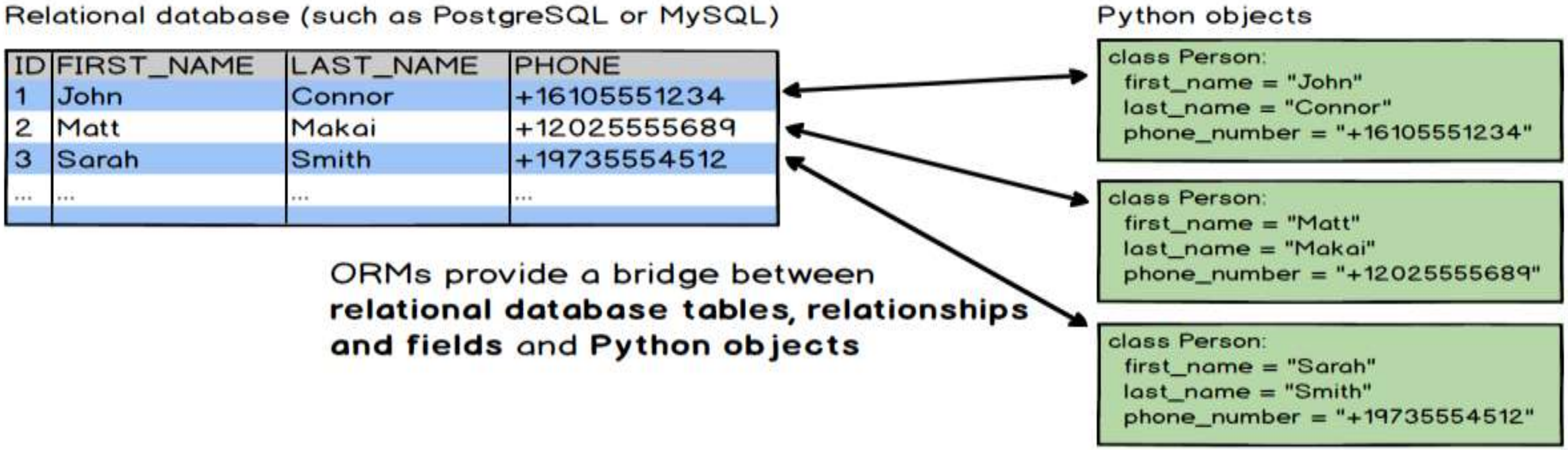
Object-Relational Managers (ORMs):

✓An object-relational mapper (ORM**) is a code library that automates the transfer of data stored in relational databases tables into objects that are more commonly used in application code**.

Object – This part represents **the objects and programming language** where the framework is used, for example **Python.**

Relational – This part represents **the RDBMS database you're using** like – MySQL, Oracle Database, PostgreSQL, MariaDB, PerconaDB, TokuDB.

Mapping – This final part represents the **bridge and connection between the two previous parts,** the objects and the database tables.



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**

✓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.**

✓For example, **without an ORM a developer** would write the following SQL statement to retrieve every row in the USERS table where the zip_code column is 94107:

```
SELECT * FROM USERS WHERE zip_code=94107;
```

✓The equivalent **Django ORM query** would instead look like the following Python code:

```
# obtain everyone in the 94107 zip code and assign to users variable
        users = Users.objects.filter(zip_code=94107)
```

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

Python Class == SQL Table

Instance of the Class == Row in the Table

✓The most well-known **Python ORMs** today are:

1. SQLAlchemy
2. Peewee
3. The Django ORM
4. PonyORM
5. SQLObject
6. Tortoise ORM

✓**SQLAlchemy** is a library that facilitates the communication between **Python programs** and **databases.**

✓ Most of the times, this library is used as an **Object Relational Mapper (ORM)** tool that translates **Python classes to tables** on relational databases and automatically converts function calls to SQL statements.

✓SQLAlchemy provides **a standard interface** that allows developers **to create database-agnostic code** to communicate with a wide variety of database engines.