# Introduction to The Python Database API Specification v2.0

The Python Database API Specification v2.0 — documented in PEP 249 — defines a common API for accessing databases with Python. The DB-API attempts to ensure that database access is consistent across different databases. Many Python-based database adapters follow the DB-API specification.

The DB-API is designed around two primary concepts: **connections and cursors**. The DB-API defines the required and optional attributes and methods for both connections and cursors.

---

Connections are responsible for establishing and maintaining a connection to the underlying database.

**Connection objects include at least the following methods and attributes.**

- .close()
    - Close the connection. Uncommitted changes are lost upon close. Using a connection after it has been closed raises an exception.
- .commit()
    - Save pending changes. May not be required for databases using auto-commit.
- .rollback()
    - Revert pending changes. May not be supported by the underlying database.
- .cursor()
    - Return a new Cursor object.

---

Cursors are responsible for interacting with the database through database operations.

**Cursor objects include at least the following methods and attributes.**

- .rowcount
    - The number of rows returned by the previous call to the execute* methods.

- .close()
    - Close the cursor. Using a cursor after it has been closed raises an exception.
- .execute(operation [, parameters])
    - Execute an operation.
- .executemany(operation, seq_of_parameters )
    - Execute an operation for each sequence of parameters.
- .fetchone()
    - Return the next row from the results of the previous call to one of the execute* methods.
    - Return None when no rows remain.
- .fetchmany([size=cursor.arraysize])
    - Return the results of the previous call to one of the execute* methods. The maximum number of returned rows is limited by the size parameter.
    - Return an empty sequence when no rows remain.
- .fetchall()
    - Return all remaining rows from the results of the previous call to one of the execute* methods.
    - Return an empty sequence when no rows remain.
- .arraysize
    - Attribute representing the default number of rows to return when calling fetchmany. Default: 1.

The following is a basic demonstration of the DB-API v2.0 using the built-in sqlite3 module. In this example an in-memory database is created and queried.

```
import sqlite3

# SQLite supports file based and in-memory databases.
# The str ':memory:' instructs SQLite to create a connection to an in-memory db.
dbconn = sqlite3.connect(':memory:')
# Cursors are created by connections.
cursor = dbconn.cursor()
# Create a database table with a single column named phrase.
# The column stores text based data.
cursor.execute('CREATE TABLE Greeting (phrase TEXT);')
```

```
# Insert two rows into the database.
cursor.execute('INSERT INTO Greeting (phrase) VALUES (?);', ('Hello',))
cursor.execute('INSERT INTO Greeting (phrase) VALUES (?);', ('Hola',))
# Save the changes to the database.
dbconn.commit()
# Query the Greeting table for all phrases.
# Once queried the results will be stored in the cursor.
cursor.execute('SELECT phrase FROM Greeting;')
# Fetch the data from the Greeting table.
assert cursor.fetchall() == [('Hello',), ('Hola',)]
# Close the connection to the database.
# Once closed no further operations will succeed.
dbconn.close()

# If the code made it this far - all assertions are accurate. :-)
################################################################################
print('No assertion errors')
```

Using a DB-API compliant database module begins with a **Connection** object. The **Connection** object is responsible for establishing and maintaining a connection to the underlying database. The type of connection (network, disk, memory, etc) depends on the implementation and database.

Commonly different database modules include some form of **connect** callable which accepts arguments and returns a **Connection** object. The connect callable of the sqlite3 module includes a positional argument specifying a database - along with optional keyword arguments.

The DB-API requires **Connection** objects to include a method named cursor used to return a new **Cursor** object. **Cursor** objects are used to execute database operations and provide access to the results.

**Cursor** objects include methods used to perform database operations and optionally return results. The methods used to perform operations are the **execute** and **executemany** methods.

The **execute** method is used to perform individual operations. The **executemany** method is used to execute a database operation for each sequence of parameters. DB-API compliant modules commonly provide a mechanism for adding parameters to the provided operation. The sqlite3 module defaults to using a question mark as the parameter replacement character.

The **commit** method of a **Connection** object saves pending transactional changes. Some databases include an **auto-commit** option. Calling **commit** performs no operation when auto-commit is enabled.

SQLite enables auto-commit by default. However, consistent use of the **commit** method after database changes — regardless of auto-commit — allows code to be more easily ported to another database with a DB-API v2.0 compliant module.

Operations passed to **execute*** methods may produce a resulting data set. Cursor objects store resulting data from the last operation. The methods used to retrieve data from the cursor are the **fetchone**, **fetchmany** and **fetchall** methods.

The **fetchall** method returns all data resulting from the previous operation. Data is returned as a sequence of sequences. The outer sequence stores rows and each inner sequence stores columnar data.

Returned rows from the **fetch*** methods are consumed and will no longer exist in the cursor. The **fetchall** method returns an empty list once the cursor is empty.

The **fetchone** method returns one row at a time until the cursor is empty. The **fetchone** method returns **None** once the cursor is empty.

The **fetchmany** method returns one row at a time by default with options for configuration.

The **Connection** object's **close** method is used to close the connection to the underlying database. Attempts to perform operations with a closed connection result in an exception.

PEP 249 specifies that **Connection** objects may include an optional **rollback** method. Databases with transaction support can use this method to rollback pending changes. This is common in cases where specific changes must all succeed or they must all fail.

## Security Considerations

Injection vulnerabilities have been one of the most common types of vulnerabilities for many years.

SQL injection is a common form of injection vulnerability which occurs when untrusted data is used to build a query string without sufficiently sanitizing and or escaping the provided data.

**Example**

The following example code dynamically builds a query string using user-supplied data.

> **NOTE**: This is demonstrating **insecure** code. Use query parameters whenever possible.

```
# Prompt a user to enter a name to lookup.
animal_name = input('Enter a name to lookup: ')
# Add the user-supplied name into the query
```

```
cursor.execute(f"SELECT Animal, Name FROM Animal WHERE Name = '{animal_name}';")
# Additional code here
...
```

Providing a valid username produces the correct results. For example specifying the name **Ada** builds the following query:

SELECT Animal, Name FROM Animal WHERE Name = 'Ada';

Providing malicious input allows an attacker to change the query. For example specifying the value **' or 1=1; --** builds the following query:

SELECT Animal, Name FROM Animal WHERE Name = '' or 1=1; --';

This where clause always evaluates as true and returns all rows. The two hyphens at the end are used to comment out the remaining query string.

The database engine treats this as a valid query; returning all rows rather than the expected individual row. SQL injection vulnerabilities range in severity depending on many factors. Database adapters commonly include a mechanism to protect against this style of attack. Parameterized queries replace placeholder characters with properly sanitized data.

> **NOTE:** The sqlite3 module uses a question mark as its default placeholder character.

Commonly user provided data is included inside database operations. Query parameters prevent SQL injection attacks by sanitizing and or escaping the provided arguments. Dynamically building a query string with untrusted data is susceptible to SQL injection attacks.

## Summary

The Python Database API Specification v2.0 -- documented in PEP 249 -- defines a common API for accessing databases with Python. The DB-API attempts to ensure that database access is consistent across different databases.

Working with DB-API compliant modules begins with a **Connection** object.

**Connection objects are responsible for:**

- Establishing and maintaining a connection to the underlying database.
- Committing pending changes.
- Rolling back pending changes.
- Closing connections.
- Creating Cursor objects.

**Connection** objects create new **Cursor** objects using the cursor method.

**Cursor objects are responsible for:**

- Executing database operations.
- Fetching data resulting from database operations.

Many database adapters for Python follow the DB-API specification.

**Examples:**

- sqlite3
- psycopg2
- pymssql
- ...

## Point of Interest

Using DB-API v2.0 compliant modules provides a consistent API regardless of the underlying database. Third-party modules – such as SQL Alchemy – commonly build on top of DB-API modules to provide higher level functionality such as object relational mappers (ORMs). ORMs enable developers to interact with relational databases using Python objects in place of hand-written queries. They typically provide an abstraction for multiple database engines. Allowing the same code to be used for multiple database engines with little or no code changes.