Chapter 24
Python DB-API

24.1 Accessing a Database from Python

The standard for accessing a database in Python is the Python DB-API. This specifies a set of standard interfaces for modules that wish to allow Python to access a specific database. The standard is described in PEP 249 (https://www.python.org/dev/peps/pep-0249)—a PEP is a Python Enhancement Proposal.

Almost all Python database access modules adhere to this standard. This means that if you are moving from one database to another, or attempting to port a Python program from one database to another, then the APIs you encounter should be very similar (although the SQL processed by different database can also differ). There are modules available for most common databases such as MySQL, Oracle, Microsoft SQL Server etc.

24.2 The DB-API

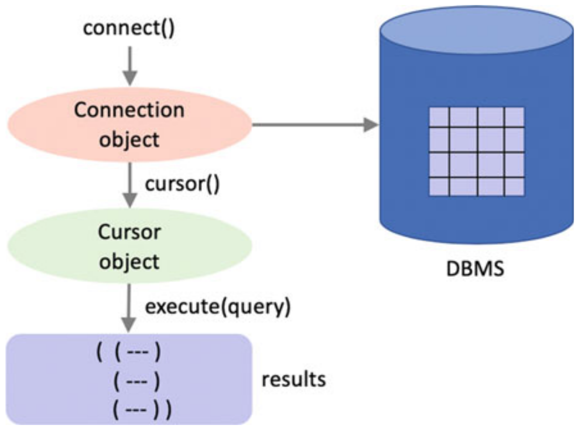
There are several key elements to the DB_API these are:

- **The connect function**. The `connect()` function that is used to connect to a database and returns a Connection Object.
- **Connection Objects**. Within the DB-API access to a database is achieved through connection objects. These connection objects provide access to cursor objects.
- **Cursor objects** are used to execute SQL statements on the database.
- **The result of an execution**. These are the results that can be fetched as a sequence of sequences (such a tuple of tuples). The standard can thus be used to select, insert or update information in the database.

© Springer Nature Switzerland AG 2019

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_24

These elements are illustrated below:



The standard specifies a set of functions and objects to be used to connect to a database. These include the `connection` function, the Connection Object and the Cursor object.

The above elements are described in more detail below.

24.2.1 The Connect Function

The connection function is defined as:

- `connect(parameters...)`

It is used to make the initial connection to the database. The connection returns a Connection Object. The parameters required by the connection function are database dependent.

24.2.2 The Connection Object

The Connection Object is returned by the `connect()` function. The Connection object provides several methods including:

- `close()` used to close the connection once you no longer need it. The connection will be unusable from this point onwards.
- `commit()` used to commit a pending transaction.

- `rollback()` used to rollback all the changes made to the database since the last transaction commit (optional as not all databases provide transaction support).
- `cursor()` returns a new `Cursor` object to use with the connection.

24.2.3 The Cursor Object

The Cursor object is returned from the `connection.cusor()` method. A Cursor Object represents a database cursor, which is used to manage the context of a fetch operation or the execution of a database command. Cursors support a variety of attributes and methods:

- `cursor.execute(operation, parameters)` Prepare and execute a database operation (such as a query statement or an update command). Parameters may be provided as a sequence or mapping and will be bound to variables in the operation. Variables are specified in a database-specific notation.
- `cursor.rowcount` a read-only attribute providing the number of rows that the last `cursor.execute()` call returned (for select style statements) or affected (for update or insert style statements).
- `cursor.description` a read only attribute providing information on the columns present in any results returned from a SELECT operation.
- `cursor.close()` closes the cursor. From this point on the cursor will not be usable.

In addition, the Cursor object also provides several fetch style methods. These methods are used to return the results of a database query. The data returned is made up of a sequence of sequences (such as a tuple of tuples) where each inner sequence represents a single row returned by the SELECT statement. The fetch methods defined by the standard are:

- `cursor.fetchone()` Fetch the next row of a query result set, returning a single sequence, or `None` when no more data is available.
- `cursor.fetchall()` Fetch all (remaining) rows of a query result, returning them as a sequence of sequences.
- `cursor.fetchman(size)` Fetch the next set of rows of a query result, returning a sequence of sequences (e.g. a tuple of tuples). An empty sequence is returned when no more rows are available. The number of rows to fetch per call is specified by the parameter.

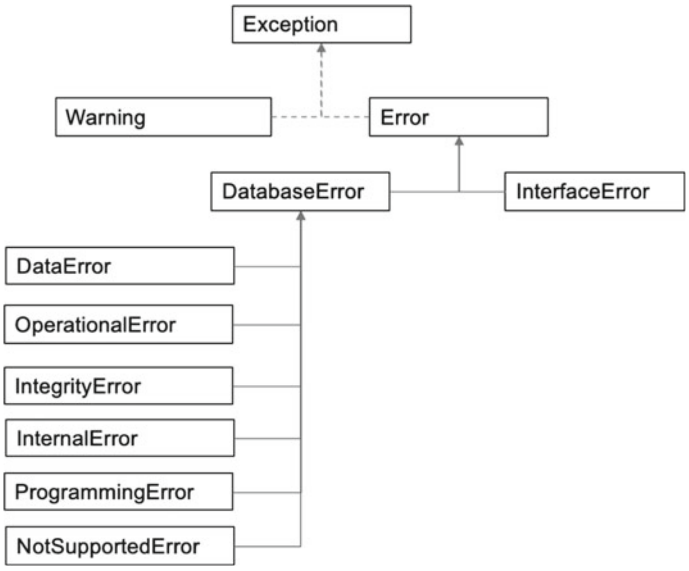
24.2.4 Mappings from Database Types to Python Types

The DB-API standard also specifies a set of mappings from the types used in a database to the types used in Python. For a full listing see the DB-API standard itself but the key mappings include:

`Date(year, month, day)`	Represents a database date
`Time(hour, minute, second)`	Represents a time database value
`Timestamp(year, month, day, hour, minute, second)`	Holds a database time stamp value
`String`	Used to represent string like database data (such as VARCHARs)

24.2.5 Generating Errors

The standard also specifies a set of Exceptions that can be thrown in different situations. These are presented below and in the following table:



The above diagram illustrates the inheritance hierarchy for the errors and warning associated with the standard. Note that the DB-API `Warning` and `Error` both extend the `Exception` class from standard Python; however, depending on the specific implementation there may be one or more additional classes in the hierarchy between these classes. For example, in the PyMySQL module there is a

MySQLError class that extends Exception and is then extended by both Warning and Error.

Also note that Warning and Error have no relationship with each other. This is because Warnings are not considered Errors and thus have a separate class hierarchies. However, the Error is the root class for all database Error classes.

A description of each Warning or Error class is provided below.

Warning	Used to warn of issues such as data truncations during inserting, etc.
Error	The base class of all other error exceptions
InterfaceError	Exception raised for errors that are related to the database interface rather than the database itself
DatabaseError	Exception raised for errors that are related to the database
DataError	Exception raised for errors that are due to problems with the data such as division by zero, numeric value out of range, etc.
OperationalError	Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, etc.
IntegrityError	Exception raised when the relational integrity of the database is affected
InternalError	Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc.
ProgrammingError	Exception raised for programming errors, e.g. table not found, syntax error in the SQL statement, wrong number of parameters specified, etc.
NotSupportedError	Exception raised in case a method or database API was used which is not supported by the database, e.g. requesting a .rollback() on a connection that does not support transactions or has transactions turned off

24.2.6 Row Descriptions

The Cursor object has an attribute description that provides a sequence of sequences; each sub sequence provides a description of one of the attributes of the data returned by a SELECT statement. The sequence describing the attribute is made up of up to seven items, these include:

- name representing the name of the attribute,
- type_code which indicates what Python type this attribute has been mapped to,
- display_size the size used to display the attribute,
- internal_size the size used internally to represent the value,

- `precision` if a real numeric value the precision supported by the attribute,
- `scale` indicates the scale of the attribute,
- `null_ok` this indicates whether null values are acceptable for this attribute.

The first two items (`name` and `type_code`) are mandatory, the other five are optional and are set to `None` if no meaningful values can be provided.

24.3 Transactions in PyMySQL

Transactions are managed in PyMySQL via the database connection object. This object provides the following method:

- `connection.commit()` this causes the current transaction to commit all the changes made permanently to the database. A new transaction is then started.
- `connection.rollback()` this causes all changes that have been made so far (but not permanently stored into the database i.e. Not committed) to be removed. A new transaction is then started.

The standard does not specify how a database interface should manage turning on and off transaction (not least because not all databases support transactions). However, MySQL does support transactions and can work in two modes; one supports the use of transactions as already described; the other uses an *auto commit* mode. In auto commit mode each command sent to the database (whether a `SELECT` statement or an `INSERT/UPDATE` statement) is treated as an independent transaction and any changes are automatically committed at the end of the statement. This *auto commit* mode can be turned on in PyMySQL using:

- `connection.autocommit(True)` turn on autocommit (`False` to turn off auto commit which is the default).

Other associated methods include

- `connection.get_autocommit()` which returns a boolean indicating whether auto commit is turned on or not.
- `connection.begin()` to explicitly begin a new transaction.

24.4 Online Resources

See the following online resources for more information on the Python Database API:

- https://www.python.org/dev/peps/pep-0249/ Python Database API Specification V2.0.
- https://wiki.python.org/moin/DatabaseProgramming Database Programming in Python.
- https://docs.python-guide.org/scenarios/db/ Databases and Python.