

Lecture 12 : Database Access

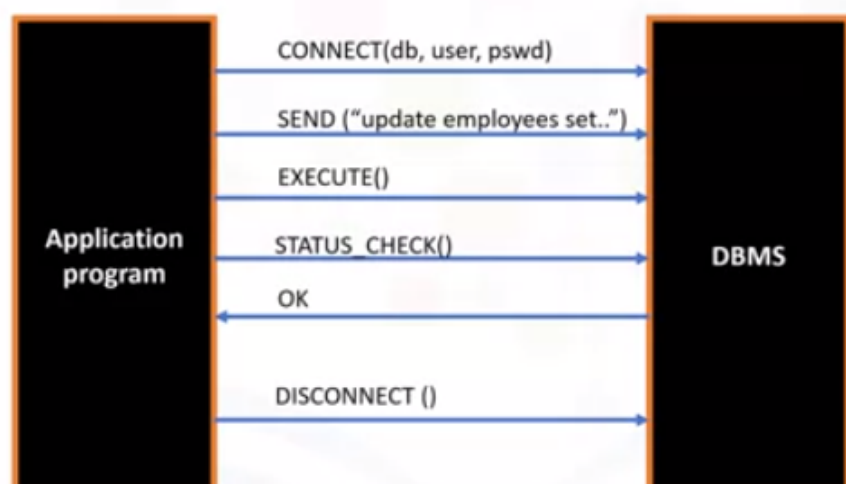
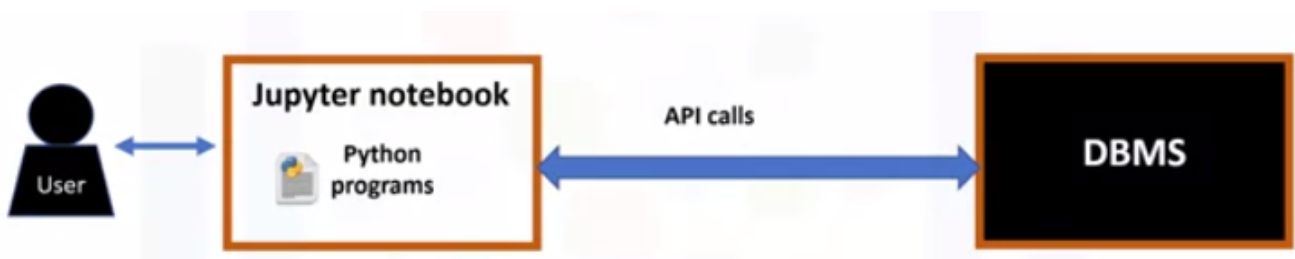
Data Science, DST, UIC

1 Python Database API (DB-API)

In this lecture we first discuss the basics of database APIs in python. As you may have learned, there are a lot of DBMS applications around, and most of them have their corresponding low-level APIs:

Application or Database	SQL API
MySQL	MySQL C API
PostgreSQL	psycopg2
IBM DB2	ibm_db
SQL Server	dblib API
Database access for Microsoft Windows OS	ODBC
Oracle	OCI
Java	JDBC

Python also provides APIs to access these different DBMS. This API is called the DB-API in python. The Python Database API (DB-API) defines a standard interface for Python database access modules, which defines how common functions like connect, close, commit, and rollback must behave. The current version of the specification is version 2.0. It's documented in PEP 249. Nearly all Python database modules such as `sqlite3`, `psycopg`, and `mysql-python` conform to this interface.



This API provides a general standard to access different DBMS, instead of writing different programs for different DBMS.

The only difference for using different DBMS in python is that you need to import different modules dedicated for that DBMS you are using:

DBMS	module
SQLite	sqlite3
MySQL	mysql
PostgreSQL	psycopg2
IBM DB2	ibm_db
SQL Server	pymssql
Oracle	pyodbc
...	...

More detailed information can be found in the following websites:

- <https://wiki.python.org/moin/DatabaseProgramming> (<https://wiki.python.org/moin/DatabaseProgramming>).
- <https://www.python.org/dev/peps/pep-0249/> (<https://www.python.org/dev/peps/pep-0249/>).

2 Examples using the DB-API

The DB-API provides a minimal standard for working with databases using Python structures and syntax wherever possible. Using API includes the following typical steps:

- Importing the API module.
- Acquiring a connection with the database.
- Issuing SQL statements and store procedures.
- Closing the connection

```
from dbmodule import connect
#Create connection object
Connection =
connect('databasename',
'username', 'pswd')
```

```
#Create a cursor object
Cursor=connection.cursor()
```

#Run Queries

```
Cursor.execute('select *
from mytable')
Results=cursor.fetchall()
```

#Free resources

```
Cursor.close()
Connection.close()
```

2.1 SQLITE

SQLite is a simple relational database system, which saves its data in regular data files or even in the internal memory of the computer, i.e. the RAM. It was developed for embedded applications, like Mozilla-Firefox (Bookmarks), Symbian OS or Android. SQLITE is "quite" fast, even though it uses a simple file. It can be used for large databases as well.

If you want to use SQLite, you have to import the module `sqlite3`. To use a database, you have to create first a `connection` object. The connection object will represent the database. The argument of `connection` - in the following example "company.db" - functions both as the **name of the file**, where the data will be stored, and as the **name of the database**. If a file with this name exists, it will be opened. It has to be a SQLite database file of course! The file does not have to exist.

```
In [1]: import sqlite3
```

```
In [2]: # Open database connection
connection = sqlite3.connect("company.db")
```

We have now created a database with the name "company". It's like having sent the command `CREATE DATABASE company;` to a SQL server. If you call `sqlite3.connect('company.db')` again, it will open the previously created database.

After having created an empty database, you will most probably add one or more tables to this database. The SQL syntax for creating a table "employee" in the database "company" looks like this:

```
CREATE TABLE employee (
    staff_number INT NOT NULL,
    fname VARCHAR(20),
    lname VARCHAR(30),
    gender CHAR(1),
    joining DATE,
    birth_date DATE,
    PRIMARY KEY (staff_number) );
```

This is the way, somebody might do it on a SQL command shell. Of course, we want to do this directly from Python. To be capable to send a command to "SQL", or SQLite, we need a `cursor` object. Usually, a cursor in SQL and databases is a control structure to traverse over the records in a database. So it's used for the fetching of the results. In SQLite (and other Python DB interfaces) it is more generally used for performing all SQL commands.

```
In [3]: cursor = connection.cursor()
```

```
In [4]: sql_command1 = """
Drop TABLE if exists employee;
"""

sql_command2="""CREATE TABLE employee(
    staff_number INTEGER PRIMARY KEY,
    fname VARCHAR(20),
    lname VARCHAR(30),
    gender CHAR(1),
    joining DATE,
    birth_date DATE);"""
```

```
In [5]: cursor.execute(sql_command1)
        cursor.execute(sql_command2)
```

```
Out[5]: <sqlite3.Cursor at 0x25b0f288030>
```

```
In [6]: try:
        sql_command = """INSERT INTO employee (staff_number, fname, lname, gender, birth_date)
                        VALUES (8, "William", "Shakespeare", "m", "1961-10-25");"""
        cursor.execute(sql_command)
    except sqlite3.IntegrityError as msg:
        print(msg)
```

```
In [7]: sql_command = """INSERT INTO employee (staff_number, fname, lname, gender, birth_date)
        VALUES (NULL, "Frank", "Schiller", "m", "1955-08-17");"""
        cursor.execute(sql_command)
```

```
Out[7]: <sqlite3.Cursor at 0x25b0f288030>
```

```
In [8]: cursor.execute("SELECT * FROM employee")
        print("fetchall:")
        result = cursor.fetchall()
        for r in result:
            print(r)
        cursor.execute("SELECT * FROM employee")
        print("\nfetch one:")
        res1 = cursor.fetchone()
        print(res1)
        res2 = cursor.fetchone()
        print(res2)
        res3 = cursor.fetchone()
        print(res3)
```

```
fetchall:
(8, 'William', 'Shakespeare', 'm', None, '1961-10-25')
(9, 'Frank', 'Schiller', 'm', None, '1955-08-17')
```

```
fetch one:
(8, 'William', 'Shakespeare', 'm', None, '1961-10-25')
(9, 'Frank', 'Schiller', 'm', None, '1955-08-17')
None
```

In SQLite, a column declared *INTEGER PRIMARY KEY* will autoincrement. There is a *ROWID* for every SQLite table whether you ask for it or not. If you include a column of type *INTEGER PRIMARY KEY*, that column points at (is an alias for) the automatic *ROWID* column. *ROWID* (by whatever name you call it) is assigned a value whenever you *INSERT* a row, as you would expect. If you explicitly assign a non-NULL value on *INSERT*, it will get that specified value instead of the auto-increment. If you explicitly assign a value of *NULL* on *INSERT*, it will get the next auto-increment value.

```
In [9]: # never forget this, if you want the changes to be saved:
        connection.commit()
        connection.close()
```

Of course, in most cases, you will not literally insert data into a SQL table. You will rather have a lot of data inside of some Python data type e.g. a dictionary or a list, which has to be used as the input of the insert statement.

The following working example, assumes that you have already an existing database company.db and a table employee. We have a list with data of persons which will be used in the *INSERT* statement:

```
In [10]: connection = sqlite3.connect("company.db")

cursor = connection.cursor()

staff_data = [ ("William", "Shakespeare", "m", "1961-10-25"),
                ("Frank", "Schiller", "m", "1955-08-17"),
                ("Jane", "Wall", "f", "1989-03-14") ]

for p in staff_data:
    format_str = """INSERT INTO employee (staff_number, fname, lname, gender, birth_date)
    VALUES (NULL, "{first}", "{last}", "{gender}", "{birthdate}");"""

    sql_command = format_str.format(first=p[0], last=p[1], gender=p[2], birthdate = p[3])
    cursor.execute(sql_command)
```

```
In [11]: cursor.execute("SELECT * FROM employee")
print("fetchall:")
result = cursor.fetchall()
for r in result:
    print(r)
cursor.execute("SELECT * FROM employee")
print("\nfetch one:")
res = cursor.fetchone()
print(res)
print("\nfetch multiple:")
res1 = cursor.fetchmany(3)
print(res1)

connection.commit()

connection.close()
```

```
fetchall:
(8, 'William', 'Shakespeare', 'm', None, '1961-10-25')
(9, 'Frank', 'Schiller', 'm', None, '1955-08-17')
(10, 'William', 'Shakespeare', 'm', None, '1961-10-25')
(11, 'Frank', 'Schiller', 'm', None, '1955-08-17')
(12, 'Jane', 'Wall', 'f', None, '1989-03-14')

fetch one:
(8, 'William', 'Shakespeare', 'm', None, '1961-10-25')

fetch multiple:
[(9, 'Frank', 'Schiller', 'm', None, '1955-08-17'), (10, 'William', 'Shakespeare', 'm', None, '1961-10-25'), (11, 'Frank', 'Schiller', 'm', None, '1955-08-17')]
```

2.2 MySQL

```
In [1]: # Use "pip install mysql-connector-python" install the module
import mysql.connector
```

Create a cursor pointing to the database we have already set up for each student. Make sure to use try-except-finally statement while connecting to the database and close the cursor and connection in finally statement.

```
In [ ]: try:
        connection = mysql.connector.connect (host = "172.31.12.170",
                                                user = "dstest", # use your username
                                                passwd = "123456", # use your password
                                                db = "dstest") # specify the database

        cursor = connection.cursor()
        cursor.execute ("SELECT VERSION()")
        row = cursor.fetchone()
        print("server version:", row[0])
    except mysql.connector.Error as msg:
        print(msg)
    except:
        print("Exception.")
    finally:
        if (connection.is_connected()):
            cursor.close()
            connection.close() # DO NOT forget to close the connection.
            print("MySQL connection is closed")
```

Everyone should have a database already:

```
In [ ]: import mysql.connector

try:
    connection = mysql.connector.connect (host = "172.31.12.170",
                                          user = "junkai",
                                          passwd = "77777777",
                                          db = "junkai")

    cursor = connection.cursor()
    cursor.execute("SHOW DATABASES")
    row = cursor.fetchall()
    for db in row:
        print(db)
except mysql.connector.Error as msg:
    print(msg)
except:
    print("Exception")
finally:
    if (connection.is_connected()):
        cursor.close()
    # connection.close() # comment close for subsequent database operation; otherwise, always remember to close the connection
```

Execute the SQL command to create a table called employee:

```
In [ ]: cursor = connection.cursor()

cursor.execute ("DROP TABLE IF EXISTS employee")
sql_command = """
CREATE TABLE employee (
staff_number INTEGER PRIMARY KEY,
fname VARCHAR(20),
lname VARCHAR(30),
gender CHAR(1),
joining DATE,
birth_date DATE);"""
cursor.execute(sql_command)
```

Insert some entries into the table,

```
In [ ]: staff_data = [ ("William", "Shakespeare", "m", "1961-10-25"),
                        ("Frank", "Schiller", "m", "1955-08-17"),
                        ("Jane", "Wall", "f", "1989-03-14"),
                        ]

for staff, p in enumerate(staff_data):
    format_str = """INSERT INTO employee (staff_number, fname, lname, gender, birth_date)
    VALUES ({staff_no}, '{first}', '{last}', '{gender}', '{birthdate}');"""

    sql_command = format_str.format(staff_no=staff, first=p[0], last=p[1], gender=p[2], birthdate=p[3])
    print(sql_command)
    cursor.execute(sql_command)

connection.commit() # save the change permanently
```

Make some queries to the table and print the result out:

```
In [ ]: cursor.execute("SELECT * FROM employee")
print(''Result of "SELECT * FROM employee":'')
result = cursor.fetchall()
for r in result:
    print(r)
```

Close the connection to the database.

```
In [ ]: cursor.close()
connection.close()
```

3 SQLAlchemy

- 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.
- SQLAlchemy provides a nice "Pythonic" way of interacting with databases. So rather than dealing with the differences between specific dialects of traditional SQL such as MySQL or PostgreSQL or Oracle, you can leverage the Pythonic framework of SQLAlchemy to streamline your workflow and more efficiently query your data.
- SQLAlchemy is designed to operate with a DB-API implementation built for a particular database. It uses **dialect system** to communicate with various types of DB-API implementations and databases. All dialects require that an appropriate DB-API driver is installed. The following are the dialects included:
 - Firebird
 - Microsoft SQL Server
 - MySQL
 - Oracle
 - PostgreSQL
 - SQLite
 - Sybase

```
In [ ]: import sqlalchemy
```

Whenever we want to use `SQLAlchemy` to interact with a database, we need to create an **Engine**. Engines, on SQLAlchemy, are used to interact with DB-API functions.

Engine class connects a Pool and Dialect together to provide a source of database *connectivity* and *behavior*. An object of Engine class is instantiated using the `create_engine()` function.

The `create_engine()` function takes the database as one argument. The database is not needed to be defined anywhere. The standard calling form has to send the **URL** as the first positional argument, usually a string that indicates database dialect and connection arguments. Using the code given below, we can create a database.

```
In [ ]: # sqlite:///<path>, as sqlite connects to local file, where <path> is relative:
engine = sqlalchemy.create_engine('sqlite:///company.db')
```

This example creates a `SQLite` engine to communicate with the database specified by the URL. Note that, creating an engine does not connect to the database instantly. This process is postponed to when it's needed (like when we submit a query, or when create/update a row in a table).

For other databases, the URL may differ, for example, The MySQL dialect uses MySQL-Python as the default DB-API (the `mysqlclient` DBAPI is a maintained fork of the MySQL-Python DBAPI that is no longer maintained). There are many MySQL DB-APIs available, including MySQL-connector-python and OurSQL:

The general syntax is the following:

```
dialect+driver://username:password@host:port/database
```

default

```
engine = create_engine('mysql://username:pwd@172.31.12.170/userdb')
```

mysqlclient (a maintained fork of MySQL-Python)

```
engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')
```

pymysql

```
engine = create_engine('mysql+pymysql://scott:tiger@localhost/foo')
```

The detailed documentation for using MySQL with sqlalchemy can be found [here](https://docs.sqlalchemy.org/en/latest/dialects/mysql.html) (<https://docs.sqlalchemy.org/en/latest/dialects/mysql.html>).

3.1 Viewing Table Details

SQLAlchemy can be used to automatically load tables from a database using something called **reflection**. Reflection is the process of reading the database and building the metadata based on that information. Metadata contains definitions of tables and associated objects such as index, view, triggers, etc.

```
In [ ]: conn = engine.connect()

# returns a collection of :class:`.Table` objects and their associated schema constructs.
metadata = sqlalchemy.MetaData(engine)
# objects for this table should be reflected from the database based on the given Metadata
employee = sqlalchemy.Table('employee', metadata, autoload=True)
```



```
In [ ]: # Print the column names
        print(employee.columns.keys())
```

3.2 Querying

Table and MetaData have already been imported.

```
In [ ]: #Equivalent to 'SELECT * FROM employee'
        query = sqlalchemy.select([employee])
```

```
In [ ]: ResultProxy = conn.execute(query)

        ResultSet = ResultProxy.fetchall()

        ResultSet
```

- ResultProxy: The object returned by the `execute()` method. It can be used in a variety of ways to get the data returned by the query.
- ResultSet: The actual data asked for in the query when using a fetch method such as `fetchall()` on a ResultProxy.

3.2 Inserting & Updating Data

```
In [ ]: #Inserting record one by one
        import datetime

        query = sqlalchemy.insert(employee).values(staff_number=3, fname="zhe", lname="xuanyuan", gender="m", birth_date=datetime.datetime(1999,1,1))
        ResultProxy = conn.execute(query)
```

```
In [ ]: query = sqlalchemy.select([employee])
        ResultProxy = conn.execute(query)

        ResultSet = ResultProxy.fetchall()

        ResultSet
```

```
In [ ]: #Inserting many records at ones
        query = sqlalchemy.insert(employee)
        values_list = [{'staff_number': '4', 'fname': 'rui', 'lname': 'meng', 'gender': 'f', 'birth_date': datetime.datetime(1999,1,2)},
                        {'staff_number': '5', 'fname': 'tian', 'lname': 'tang', 'gender': 'm', 'birth_date': datetime.datetime(1999,1,3)},
                        {'staff_number': '6', 'fname': 'jie', 'lname': 'li', 'gender': 'f', 'birth_date': datetime.datetime(1999,1,4)}]
        ResultProxy = conn.execute(query, values_list)
```

```
In [ ]: query = sqlalchemy.select([employee])
        ResultProxy = conn.execute(query)

        ResultSet = ResultProxy.fetchall()

        ResultSet
```

Turn the results into Pandas data frames

```
In [ ]: import pandas as pd

df = pd.DataFrame(ResultSet)
df.columns = ResultSet[0].keys()
df.head(5)
```

```
In [ ]: # Build a statement to update the birthday of xyz to 1998
query = sqlalchemy.update(employee).values(birth_date = datetime.datetime(1998, 1, 1))
query = query.where(employee.columns.lname == "xuanyuan")
results = conn.execute(query)
```

```
In [ ]: results = conn.execute(sqlalchemy.select([employee])).fetchall()
df = pd.DataFrame(results)
df.columns = results[0].keys()
df.head(5)
```

3.3 Filtering data

Lets see some examples of raw SQLite Queries and queries using SQLAlchemy.

- **where**

SQL :

```
SELECT * FROM employee
WHERE gender = f
```

SQLAlchemy :

```
In [ ]: query = sqlalchemy.select([employee]).where(employee.columns.gender == 'f')
resultsProxy = conn.execute(query)
results = resultsProxy.fetchall()
results
```

- **in**

SQL :

```
SELECT fname, lname
FROM employee
WHERE gender IN (f, m)
```

SQLAlchemy :

```
In [ ]: query = sqlalchemy.select([employee.columns.fname, employee.columns.lname]).where(employee.columns.gender.in_(['f', 'm']))
resultsProxy = conn.execute(query)
results = resultsProxy.fetchall()
results
```

- **and, or, not**

SQL :

```
SELECT * FROM employee
WHERE fname = 'zhe' AND NOT gender = 'f'
```

SQLAlchemy :

```
In [ ]: query = sqlalchemy.select([employee]).where(sqlalchemy.and_(employee.columns.fname=='zhe', employee.columns.gender!='f'))
resultsProxy = conn.execute(query)
results = resultsProxy.fetchall()
results
```

- **order by**

SQL :

```
SELECT * FROM employee
ORDER BY birth_date DESC
```

SQLAlchemy :

```
In [ ]: query = sqlalchemy.select([employee]).order_by(sqlalchemy.desc(employee.columns.birth_date))
resultsProxy = conn.execute(query)
results = resultsProxy.fetchall()
results
```

- **functions**

SQL :

```
SELECT COUNT(staff_number) FROM employee
```

SQLAlchemy :

```
In [ ]: query = sqlalchemy.select([sqlalchemy.func.count(employee.columns.staff_number)])
resultsProxy = conn.execute(query)
results = resultsProxy.fetchall()
results
```

other functions include sum, min, max, etc...

- **Join**

```
In [ ]: db = sqlalchemy.create_engine('sqlite:///joindemo.db')

metadata = sqlalchemy.MetaData(db)
```

```
In [ ]: conn = db.connect()
from sqlalchemy import inspect

# create two tables, one is users, the other is emails
users = sqlalchemy.Table('users', metadata,
    sqlalchemy.Column('user_id', sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column('name', sqlalchemy.String(40)),
    sqlalchemy.Column('age', sqlalchemy.Integer),
    extend_existing=True)

# Inspect - Get Database Information
# check whether the table already exists
inspector = inspect(db)
if 'users' in inspector.get_table_names():
    users.drop()

users.create()

emails = sqlalchemy.Table('emails', metadata,
    sqlalchemy.Column('email_id', sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column('address', sqlalchemy.String),
    sqlalchemy.Column('user_id', sqlalchemy.Integer, sqlalchemy.ForeignKey('users.user_id')),
    extend_existing=True)

if 'emails' in inspector.get_table_names():
    emails.drop()

emails.create()
```

```
In [ ]: i = users.insert()
i.execute(
    {'name': 'Mary', 'age': 30},
    {'name': 'John', 'age': 42},
    {'name': 'Susan', 'age': 57},
    {'name': 'Carl', 'age': 33}
)

j = emails.insert()
j.execute(
    # There's a better way to do this, but we haven't gotten there yet
    {'address': 'mary@example.com', 'user_id': 1},
    {'address': 'john@nowhere.net', 'user_id': 2},
    {'address': 'john@example.org', 'user_id': 3},
    {'address': 'carl@nospam.net', 'user_id': 4},
)
```

Join the two tables

```
In [ ]: def run(stmt):
    rs = stmt.execute()
    for row in rs:
        print(row)

s = sqlalchemy.select([users, emails])
run(s)
# conn = engine.connect()
# print(conn.execute(s).fetchall())
```

This will return more results than you are probably expecting. The reason is because you specified no WHERE clause, so a full join was performed, which returns every possible combination of records from tables A and B. With an appropriate WHERE clause, you'll get the restricted record set you really wanted.

```
In [ ]: s = sqlalchemy.select([users, emails], emails.c.user_id == users.c.user_id)
run(s)
```

If you're interested in only a few columns, then specify them explicitly

```
In [ ]: s = sqlalchemy.select([users.c.name, emails.c.address],
                               emails.c.user_id == users.c.user_id)
run(s)
```

There are also "smart" join objects that can figure out the correct join conditions based on the tables' foreign keys

```
In [ ]: s = sqlalchemy.join(users, emails).select()
run(s)
```

If you want all the users, whether or not they have an email address, then you want an "outer" join.

```
In [ ]: s = sqlalchemy.outerjoin(users, emails).select()
run(s)
```

Order of outer joins is important! Default is a "left outer join", which means "all records from the left-hand table, plus their corresponding values from the right-hand table, if any". Notice how this time, Susan's name will *not* appear in the results.

```
In [ ]: s = sqlalchemy.outerjoin(emails, users).select()
run(s)
```

```
In [ ]: #close the connection and dispose engine
conn.close()
engine.dispose()
```

3.4 DB-API with SQLAlchemy

Besides the aforementioned 'Pythonic' way of interacting with databases, SQLAlchemy also supports the DB-API.

```
In [ ]: engine = sqlalchemy.create_engine('sqlite:///joindemo.db')
conn = engine.connect()
result = conn.execute('SELECT * FROM users')
for r in result:
    print(r)
conn.close()
engine.dispose()
```