DATABASE SYSTEMS ASSIGNMENT4

PostgreSQL with Python

PostgreSQL with Python

- psycopg2 라이브러리를 사용하여 PosgreSQL에 연결 가능
- The main entry points of Psycopg are:
- The function connect() creates a new database session and returns a new connection instance.
- The class connection encapsulates a database session. It allows to:
    - create new cursor instances using the cursor() method to execute database commands and queries,
    - terminate transactions using the methods commit() or rollback().
- The class cursor allows interaction with the database:
    - send commands to the database using methods such as execute() and executemany(),
    - retrieve data from the database by iteration or using methods such as fetchone(), fetchmany(), fetchall()

```
import psycopg2

# Connect to an existing database
>>> conn = psycopg2.connect("dbname=test user=postgres")
```

PostgreSQL with Python

- psycopg2 라이브러리를 사용하여 PosgreSQL에 연결 가능
  - commit()
  - Commit any pending transaction to the database.
  - rollback()
    - Roll back to the start of any pending transaction. Closing a connection without committing the changes first will cause an implicit rollback to be performed.
  - close()
    - Close the connection now (rather than whenever del is executed).

```
# Open a cursor to perform database operations
>>> cur = conn.cursor()

# Execute a command: this creates a new table
>>> cur.execute("CREATE TABLE test (id serial PRIMARY KEY, num integer, data varchar);")

# Pass data to fill a query placeholders and let Psycopg perform
# the correct conversion (no more SQL injections!)
>>> cur.execute("INSERT INTO test (num, data) VALUES (%s, %s)",
...      (100, "abc'def"))

# Query the database and obtain data as Python objects
>>> cur.execute("SELECT * FROM test;")
>>> cur.fetchone()
(1, 100, "abc'def")

# Make the changes to the database persistent
>>> conn.commit()

# Close communication with the database
>>> cur.close()
>>> conn.close()
```

```
conn = psycopg2.connect(DSN)
try:
    # connection usage
finally:
    conn.close()
```

PostgreSQL with Python

- The function connect() creates a new database session and returns a new connection instance.
  - dbname – the database name (database is a deprecated alias)
  - user – user name used to authenticate
  - password – password used to authenticate
  - host – database host address (defaults to UNIX socket if not provided)
  - port – connection port number (defaults to 5432 if not provided)
- The class connection encapsulates a database session. It allows to:
  - create new cursor instances using the cursor() method to execute database commands and queries,
  - terminate transactions using the methods commit() or rollback().
- The class cursor allows interaction with the database:
  - send commands to the database using methods such as execute() and executemany(),
  - retrieve data from the database by iteration or using methods such as fetchone(), fetchmany(), fetchall().

```
conn = psycopg2.connect("dbname=test user=postgres password=secret")
conn = psycopg2.connect(dbname="test", user="postgres", password="secret")
```

```
>>> nums = ((1,), (5,), (10,))
>>> cur.executemany("INSERT INTO test (num) VALUES (%s)", nums)

>>> tuples = ((123, "foo"), (42, "bar"), (23, "baz"))
>>> cur.executemany("INSERT INTO test (num, data) VALUES (%s, %s)", tuples)
```

```
>>> cur.execute("SELECT * FROM test WHERE id = %s", (3,))
>>> cur.fetchone()
(3, 42, 'bar')

>>> cur.execute("SELECT * FROM test;")
>>> cur.fetchmany(2)
[(1, 100, "abc'def"), (2, None, 'dada')]
>>> cur.fetchmany(2)
[(3, 42, 'bar')]
>>> cur.fetchmany(2)
[]

>>> cur.execute("SELECT * FROM test;")
>>> cur.fetchall()
[(1, 100, "abc'def"), (2, None, 'dada'), (3, 42, 'bar')]
```

PostgreSQL with Python

- Passing parameters to an SQL statement happens in functions such as cursor.execute() by using %s placeholders in the SQL statement, and passing a sequence of values as the second argument of the function.

```
>>> cur.execute("""
...     INSERT INTO some_table (an_int, a_date, a_string)
...     VALUES (%s, %s, %s);
...     """,
...     (10, datetime.date(2005, 11, 18), "O'Reilly"))
```

Converted into

```
INSERT INTO some_table (an_int, a_date, a_string)
VALUES (10, '2005-11-18', 'O''Reilly');
```

- Named arguments are supported too using %(name)s placeholders in the query and specifying the values into a mapping. Using named arguments allows to specify the values in any order and to repeat the same value in several places in the query

```
>>> cur.execute("""
...     INSERT INTO some_table (an_int, a_date, another_date, a_string)
...     VALUES (%(int)s, %(date)s, %(date)s, %(str)s);
...     """,
...     {'int': 10, 'str': "O'Reilly", 'date': datetime.date(2005, 11, 18)})
```

- When parameters are used, in order to include a literal % in the query you can use the %% string:

```
>>> cur.execute("SELECT (%s % 2) = 0 AS even", (10,))       # WRONG
>>> cur.execute("SELECT (%s %% 2) = 0 AS even", (10,))      # correct
```

PostgreSQL with Python

- The Python string operator % must not be used: the execute() method accepts a tuple or dictionary of values as second parameter. Never use % or + to merge values into queries:

```
>>> cur.execute("INSERT INTO numbers VALUES (%s, %s)" % (10, 20)) # WRONG
>>> cur.execute("INSERT INTO numbers VALUES (%s, %s)", (10, 20))  # correct
```

- For positional variables binding, the second argument must always be a sequence, even if it contains a single variable (remember that Python requires a comma to create a single element tuple):

```
>>> cur.execute("INSERT INTO foo VALUES (%s)", "bar")    # WRONG
>>> cur.execute("INSERT INTO foo VALUES (%s)", ("bar"))  # WRONG
>>> cur.execute("INSERT INTO foo VALUES (%s)", ("bar",)) # correct
>>> cur.execute("INSERT INTO foo VALUES (%s)", ["bar"])  # correct
```

- The variables placeholder must always be a %s, even if a different placeholder (such as a %d for integers or %f for floats) may look more appropriate:

```
>>> cur.execute("INSERT INTO numbers VALUES (%d)", (10,))   # WRONG
>>> cur.execute("INSERT INTO numbers VALUES (%s)", (10,))   # correct
```