>>

Psycopg2

^ >>

❖ Psycopg2

Psycopg2 is a Python module that provides

- a method to connect to PostgreSQL databases
- a collection of DB-related exceptions
- a collection of type and object constructors

In order to use Psycopg2 in a Python program

```
import psycopg2
```

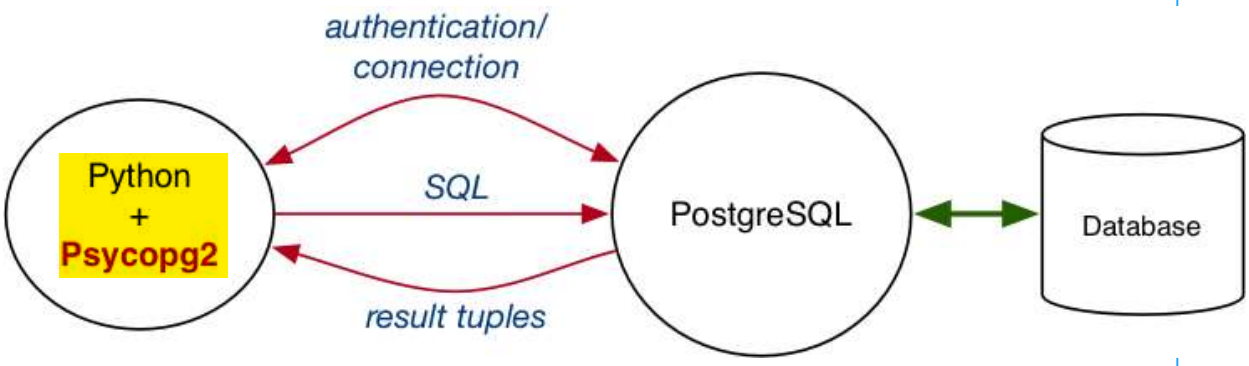
Note:

- assumes that the **psycopg2** module is installed on your system
- Psycopg2 is installed on Grieg; installing on a Mac is relatively easy

<< ^ >>

❖ Psycopg2 (cont)

Where **psycopg2** fits in the PL/DB architecture



<< ^ >>

❖ Database `connections`

`conn = psycopg2.connect(`*DB_connection_string*`)`

- creates a **`connection`** object on a named database
- effectively starts a session with the database (cf **`psql`**)
- returns a **`connection`** object used to access DB
- if can't connect, raises an exception

DB connection string components

- **`dbname`** ... name of database
- **`user`** ... user name (for authentication)
- **`password`** ... user password (for authentication)
- **`host`** ... where is the server running (default=localhost)
- **`port`** ... which port is server listening on (default=5432)

On Grieg, only **`dbname`** is required.

<< ^ >>

❖ Example: connecting to a database

Simple script that connects and then closes connection

```
import psycopg2

try:
    conn = psycopg2.connect("dbname=mydb")
    print(conn)  # state of connection after opening
    conn.close()
    print(conn)  # state of connection after closing
except Exception as e:
    print("Unable to connect to the database")
```

which, if **mydb** is accessible, produces output:

```
$ python3 ex1.py
<connection object at 0xf67186ec; dsn: 'dbname=mydb', closed: 0>
<connection object at 0xf67186ec; dsn: 'dbname=mydb', closed: 1>
```

<< ^ >>

❖ Example: connecting to a database (cont)

Example: change the script to get databaase name from command line

```
import sys
import psycopg2

if len(sys.argv) < 2:
    print("Usage: opendb DBname")
    exit(1)
db = sys.argv[1]
try:
    conn = psycopg2.connect("dbname="+db)
    print(conn)
    conn.close()
    print(conn)
except Exception as e:
    print(f"Unable to connect to database {db}")
```

<< ^ >>

❖ Operations on `connections`

`cur = conn.cursor()`

- set up a handle for performing queries/updates on database
- must create a **`cursor`** before performing any DB operations

`conn.close()`

- close the database connection **`conn`**

`conn.commit()`

- commit changes made to database since last **`commit()`**

Plus many others ... see Psycopg2 documentation

<< ^ >>

❖ Database Cursors

Cursors are "pipelines" to the database

Cursor objects allow you to ...

- execute queries, perform updates, change meta-data

Cursors are created from a database **connection**

- can create multiple cursors from the same connection
- each cursor handles one DB operation at a time
- but cursors are not isolated (can see each others' changes)

To set up a **cursor** object called **cur** ...

```
cur = conn.cursor()
```

<< ^ >>

❖ Operations on cursors

`cur.execute(`*SQL_statement*`,` *Values*`)`

- if supplied, insert values into the SQL statement
- then execute the SQL statement
- results are available via the cursor's fetch methods

Examples:

```
# run a fixed query
cur.execute("select * from R where x = 1");

# run a query with values inserted
cur.execute("select * from R where x = %s", (1,))
cur.execute("select * from R where x = %s", [1])

# run a query stored in a variable
query = "select * from Students where name ilike %s"
pattern = "%mith%"
cur.execute(query, [pattern])
```

<< ^ >>

❖ Operations on `cursors` (cont)

`cur.mogrify(` *SQL_statement, Values* **`)`**

- return the SQL statement as a string, with values inserted
- useful for checking whether **`execute()`** is doing what you want

Examples:

```
query = "select * from R where x = %s"
print(cur.mogrify(query, [1]))
Produces: b'select * from R where x = 1'

query = "select * from R where x = %s and y = %s"
print(cur.mogrify(query, [1,5]))
Produces: b'select * from R where x = 1 and y = 5'

query = "select * from Students where name ilike %s"
pattern = "%mith%"
print(cur.mogrify(query, [pattern]))
Produces: b"select * from Students where name ilike '%mith%'"

query = "select * from Students where family = %s"
fname = "O'Reilly"
print(cur.mogrify(query, [fname]))
Produces: b"select * from Students where family = 'O''Reilly'"
```

<< ^ >>

❖ Operations on `cursors` (cont)

```
list = cur.fetchall()
```

- gets all answers for a query and stores in a list of tuples
- can iterate through list of results using Python's **`for`**

Example:

```
# table R contains (1,2), (2,1), ...

cur.execute("select * from R")
for tup in cur.fetchall():
    x,y = tup
    print(x,y)    # or print(tup[0],tup[1])

# prints
1 2
2 1
...
```

<< ^ >>

❖ Operations on `cursors` (cont)

```
tup = cur.fetchone()
```

- gets next result for a query and stores in a tuple
- can iterate through list of results using Python's **`while`**

Example:

```
# table R contains (1,2), (2,1), ...

cur.execute("select * from R")
while True:
    t = cur.fetchone()
    if t == None:
        break
    x,y = tup
    print(x,y)

# prints
1 2
2 1
...
```

<< ^

❖ Operations on `cursors` (cont)

`tup = cur.fetchmany(`*nTuples*`)`

- gets next *nTuples* results for a query
- stores tuples in a list
- when no results left, returns empty list

Example:

```
# table R contains (1,2), (2,1), ...

cur.execute("select * from R")
while True:
    tups = cur.fetchmany(3)
    if tups == []:
        break
    for tup in tups:
        x,y = tup
        print(x,y)

# prints
1 2
2 1
...
```

Produced: 26 Oct 2020