Postgres and Python for the visualization of Spatial Data

Advanced Topics in Databases

Psycopg – PostgreSQL database adapter for Python

- Psycopg is the most popular PostgreSQL database adapter for the Python programming language.
- Its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety (several threads can share the same connection).
- It was designed for heavily multi-threaded applications that create and destroy lots of cursors and make a large number of concurrent INSERTs or UPDATEs.
- Psycopg 2 is mostly implemented in C as a libpq wrapper, resulting in being both efficient and secure.
- Many Python types are supported out-of-the-box and adapted to matching PostgreSQL data types;

The psycopg2 module content

- The module interface respects the standard defined in the DB API 2.0.

  psycopg2.connect(dsn=None, connection_factory=None, cursor_factory=None, async=False, **kwargs)

- Create a new database session and return a new connection object.
- The connection parameters can be specified as a libpq connection string using the dsn parameter:

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

- or using a set of keyword arguments:

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

connect()

- The basic connection parameters are:
  - 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 connection class

- Handles the connection to a PostgreSQL database instance. It encapsulates a database session.

- Connections are created using the factory function connect().

- Connections are thread safe and can be shared among many threads. See Thread and process safety for details.

connection.cursor() method

cursor(name=None, cursor_factory=None, scrollable=None, withhold=False)

- Return a new cursor object using the connection.
- If name is specified, the returned cursor will be a server side cursor (also known as named cursor). Otherwise it will be a regular client side cursor.

connection.commit(), connection.rollback(), connection.close()

commit()

- Commit any pending transaction to the database.
- By default, Psycopg opens a transaction before executing the first command: if commit() is not called, the effect of any data manipulation will be lost.
- The connection can be also set in “autocommit” mode: no transaction is automatically open, commands have immediate effect.

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. The connection will be unusable from this point forward

The cursor class

- Allows Python code to execute PostgreSQL command in a database session. Cursors are created by the connection.cursor() method: they are bound to the connection for the entire lifetime and all the commands are executed in the context of the database session wrapped by the connection.

- Cursors created from the same connection are not isolated, i.e., any changes done to the database by a cursor are immediately visible by the other cursors. Cursors created from different connections can or can not be isolated, depending on the connections’ isolation level.

close() method

- Close the cursor now. The cursor will be unusable from this point forward; an InterfaceError will be raised if any operation is attempted with the cursor.

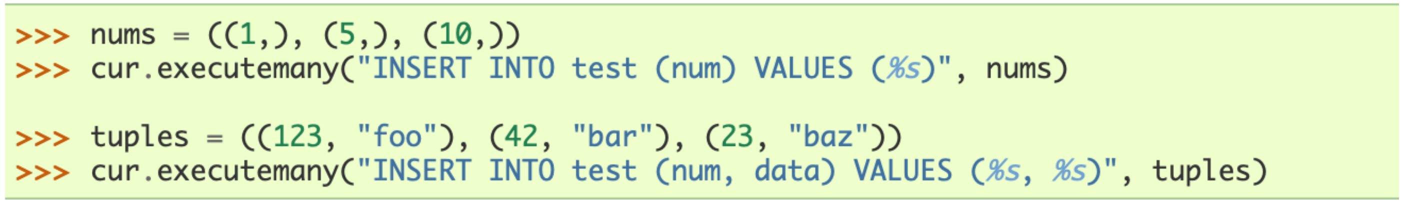
Commands execution methods

execute(query, vars=None)

- Execute a database operation (query or command).
- Parameters may be provided as sequence or mapping and will be bound to variables in the operation. Variables are specified either with positional (%s) or named (%(name)s) placeholders.
- The method returns None. If a query was executed, the returned values can be retrieved using fetch*() methods.

executemany(query, vars_list)

- Execute a database operation (query or command) against all parameter tuples or mappings found in the sequence vars_list.
- The function is mostly useful for commands that update the database: any result set returned by the query is discarded.
- Parameters are bounded to the query using the same rules described in the execute() method.



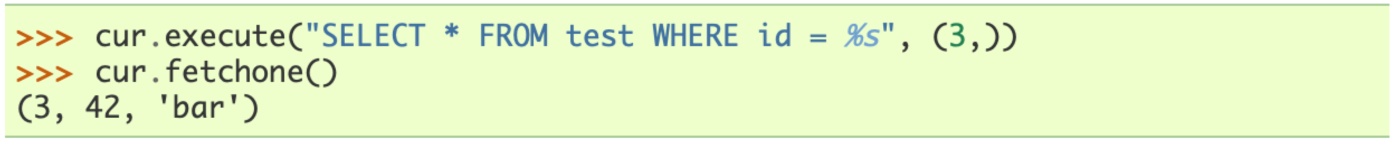
```
>>> nums = ((1,), (5,), (10,))
>>> cur.executemany("INSERT INTO test (num) VALUES (%s)", nums)

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

Results retrieval methods

- The following methods are used to read data from the database after an execute() call:

fetchone()

- Fetch the next row of a query result set, returning a single tuple, or None when no more data is available:



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

- A programming error is raised if the previous call to execute*() did not produce any result set or no call was issued yet.

Results retrieval methods

fetchall()

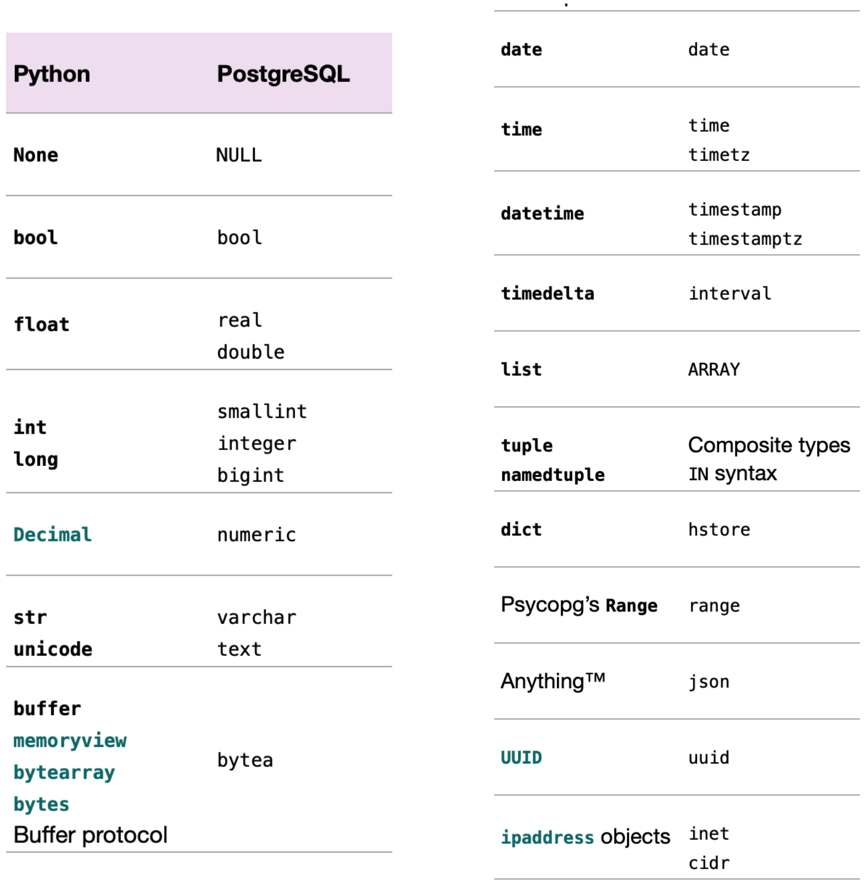
- Fetch all (remaining) rows of a query result, returning them as a list of tuples. An empty list is returned if there is no more record to fetch.

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

- A programming error is raised if the previous call to execute*() did not produce any result set or no call was issued yet.

Adaptation of Python values to SQL types

- Many standard Python types are adapted into SQL and returned as Python objects when a query is executed.
- The following table shows the default mapping between Python and PostgreSQL types:



Python	PostgreSQL
`None`	`NULL`
`bool`	`bool`
`float`	`real` `double`

`int`
`long`
	`smallint`
	`integer`
	`bigint`

`Decimal`	`numeric`
`str` `unicode`	`varchar` `text`

`buffer`
`memoryview`
`bytearray`
`bytes`
Buffer protocol
	`bytea`

`date`	`date`
`time`	`time` `timetz`
`datetime`	`timestamp` `timestamptz`
`timedelta`	`interval`
`list`	`ARRAY`
`tuple` `namedtuple`	Composite types IN syntax
`dict`	`hstore`
Psycopg's `Range`	`range`
Anything™	`json`
`UUID`	`uuid`
`ipaddress` objects	`inet` `cidr`

Further study

https://www.psycopg.org/docs/index.html

Visualization of spatial data with Matplotlib and Python

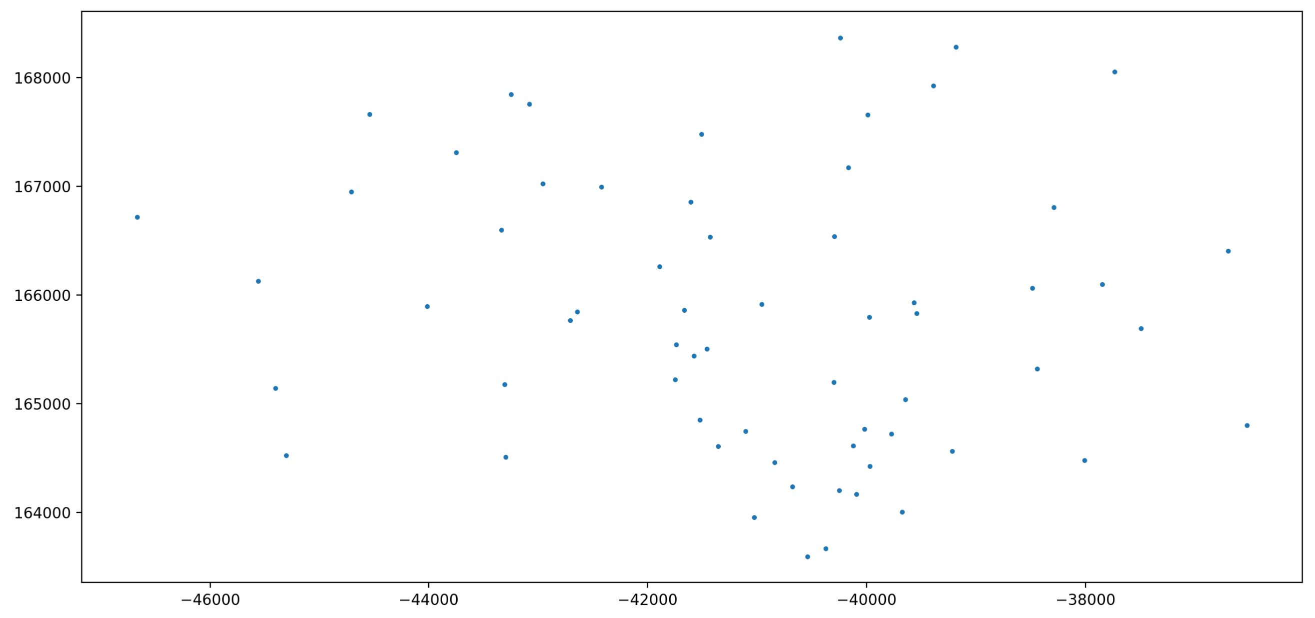
- Matplotlib is a powerful and very popular data visualization library in Python.
- Documentation and examples available at:

  https://matplotlib.org/stable/index.html

Creating a scatter plot to show the taxi stands of Porto

```
import matplotlib.pyplot as plt
import psycopg2

scale=1/30000
conn = psycopg2.connect("dbname=michelferreira user=michelferreira")
cursor_psql = conn.cursor()
sql = "select st_astext(proj_location) from taxi_stands"
cursor_psql.execute(sql)
results = cursor_psql.fetchall()
xs = []
ys = []
for row in results:
    point_string = row[0]
    point_string = point_string[6:-1]
    (x,y) = point_string.split()
    xs.append(float(x))
    ys.append(float(y))
width_in_inches = ((max(xs)-min(xs))/0.0254)*1.1
height_in_inches = ((max(ys)-min(ys))/0.0254)*1.1
fig = plt.figure(figsize=(width_in_inches*scale,height_in_inches*scale))
plt.scatter(xs,ys,s=5)
plt.show()
```

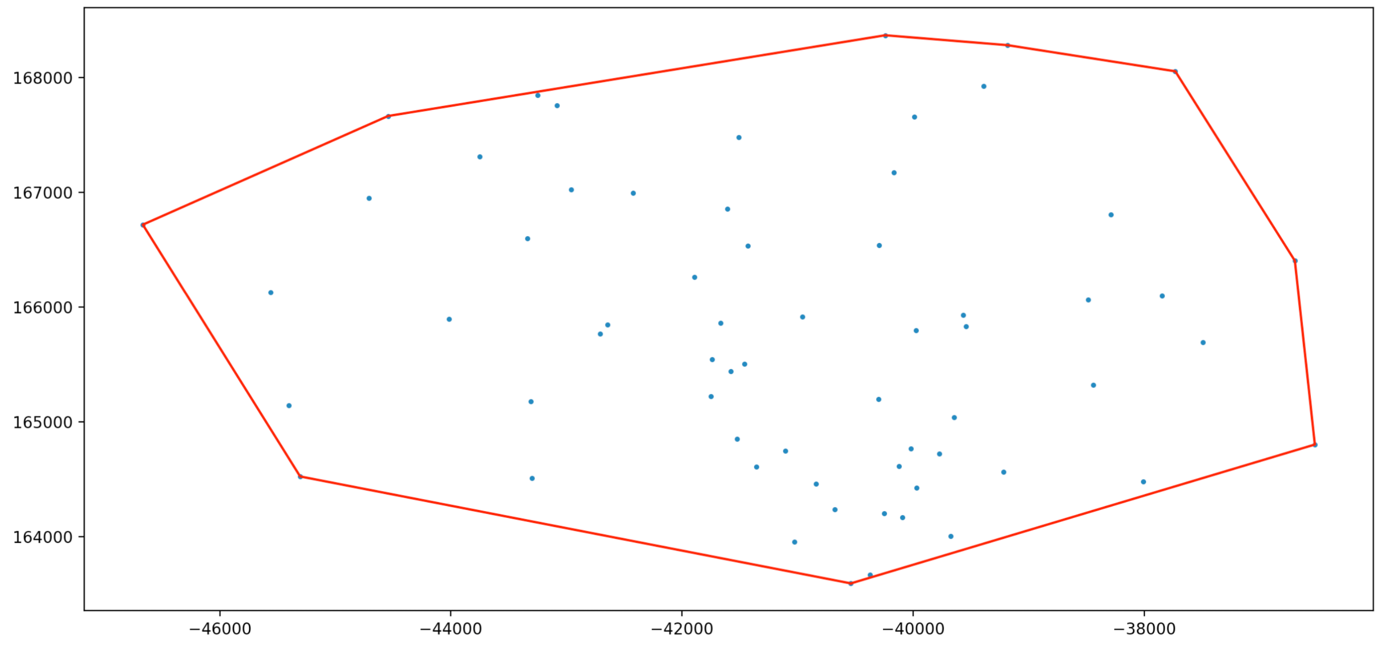
Creating a scatter plot to show the taxi stands of Porto



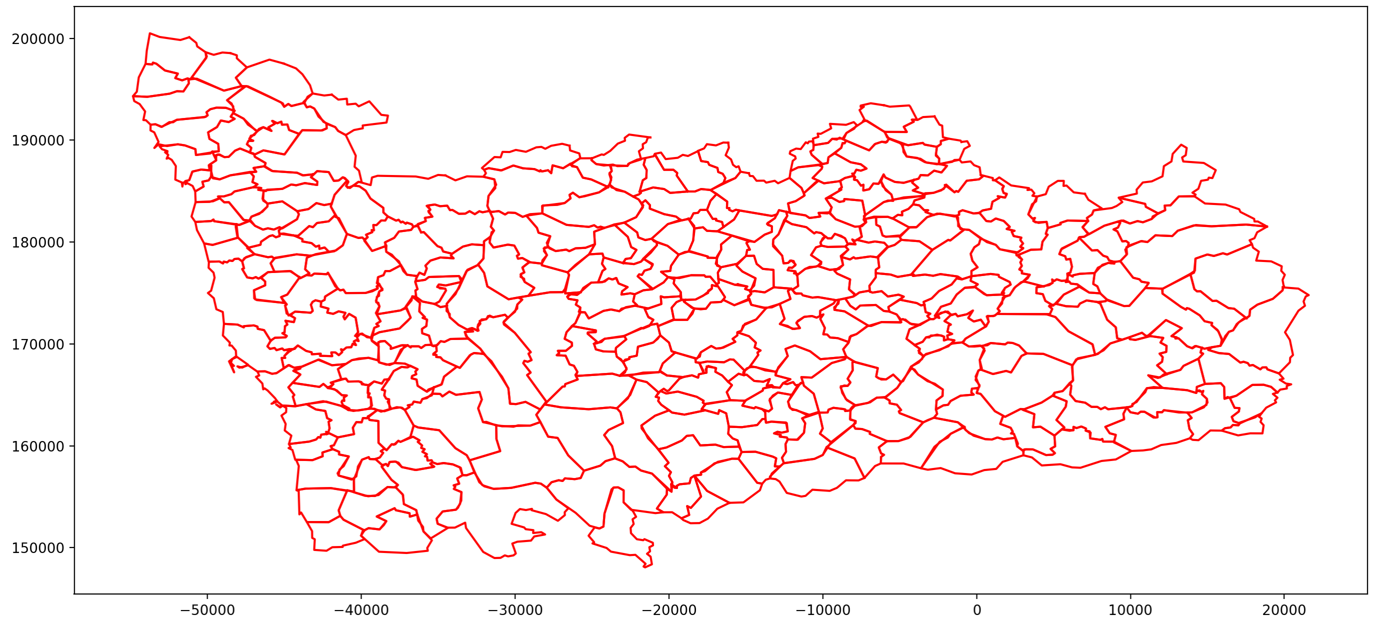
Creating a line plot with the boundary of the convex hull of taxi stands

```
sql = "select st_astext(st_convexhull(st_collect(proj_location))) from taxi_stands"
cursor_psql.execute(sql)
results = cursor_psql.fetchall()

xs = []
ys = []

for row in results:
    print(row[0])
    points_string = row[0]
    points_string = points_string[9:-2]
    points = points_string.split(',')
    for point in points:
        (x,y) = point.split()
        xs.append(float(x))
        ys.append(float(y))
        print(x,y)

plt.plot(xs,ys, color='red')
plt.show()
```

Creating a line plot with the boundary of the convex hull of taxi stands



Visualizing the parishes of the district of Porto



200000
190000
180000
170000
160000
150000
−50000
−40000
−30000
−20000
−10000
0
10000
20000