COMP5310: Principles of Data Science

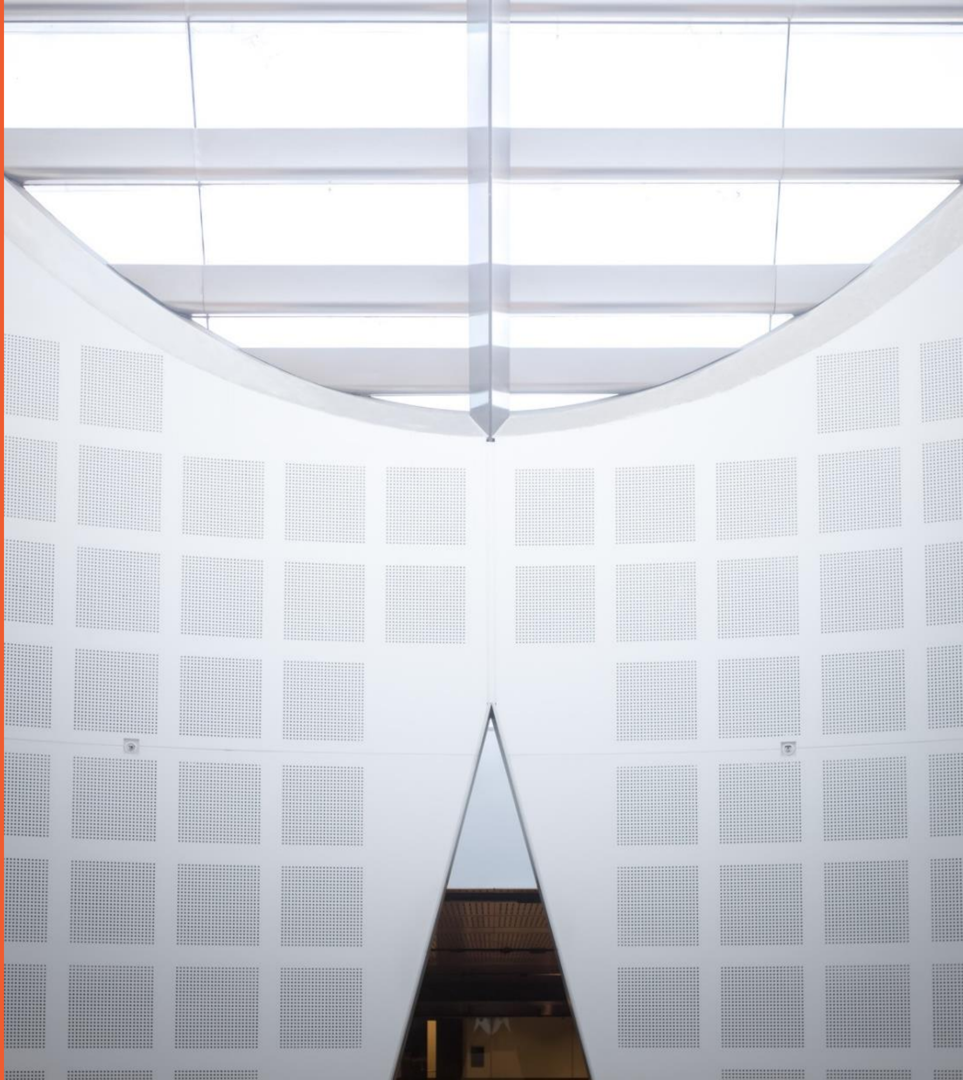
W5: Querying and Summarising Data with SQL

Presented by

Maryam Khanian

School of Computer Science

Based on slides by previous lecturers of this unit of study



Last week: Data transformation and storage with Python and SQL

Objective

- Use Python and PostgreSQL to extract, clean, transform and store data.

Lecture

- DB Access from Python.
- Data cleaning and preprocessing.
- Data Modeling and DB Creation.
- Data Loading/Storage.

Readings

- Data Science from Scratch: Ch 24

Exercises

- Python/Jupyter to load data.
- Psycopg2.
- PostgreSQL to store data.

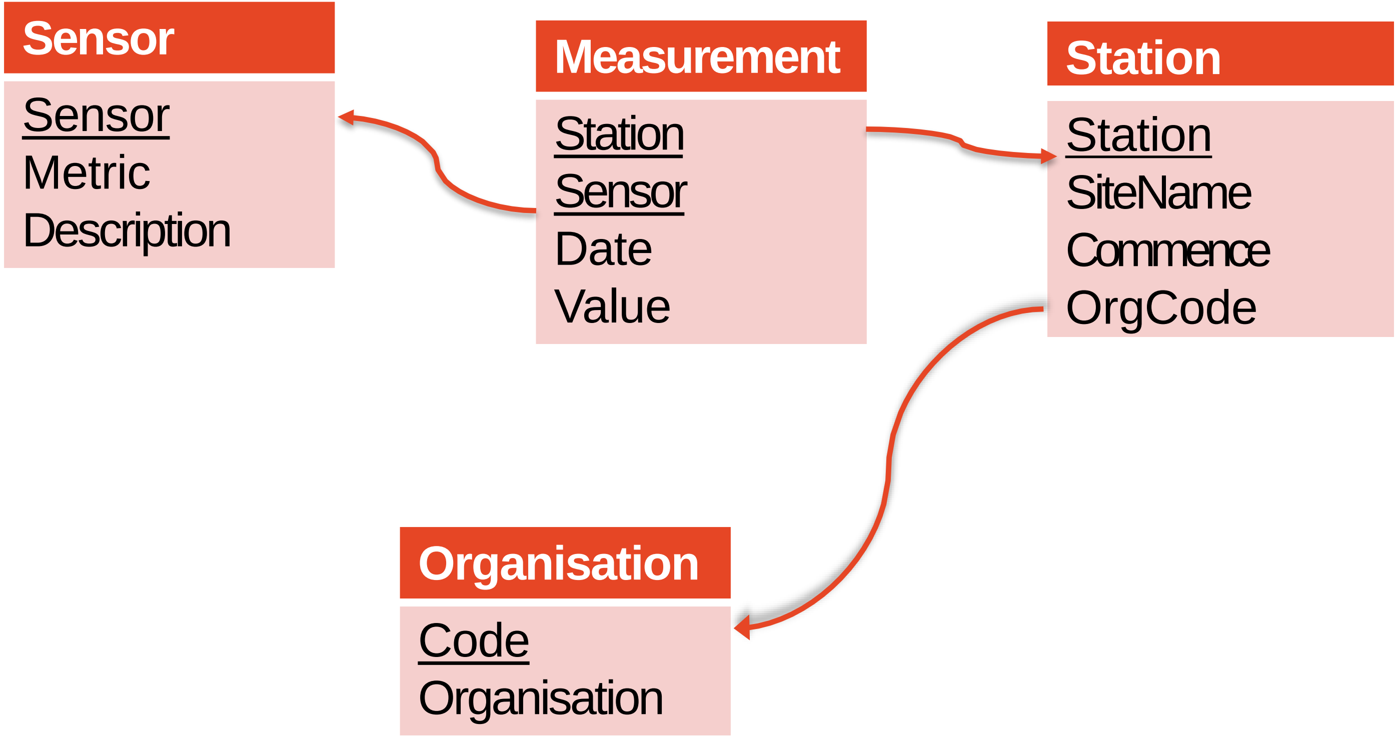
TO-DO in W4

- Ed Lessons Python modules 10-12.
- Ed Lessons SQL modules 20-21.

This week

- This week we continue where we left off last time: loading and storing data in a relational database.
  - Last week we focused on ETL: Extract-Transform-Load.
  - This week we take a clean database as given and concentrate on exploring and querying it.

Water database schema



Sensor

Sensor
Metric
Description

Measurement

Station
Sensor
Date
Value

Station

Station
SiteName
Commence
OrgCode

Organisation

Code
Organisation

QUERYING DATA WITH SQL

SELECT statement

- The **SELECT** statement retrieves data (rows) from one or more tables that fulfill a search condition.
- Clauses:
  - **SELECT:** Lists the attributes (and expressions) that should be returned from the query.
  - **FROM:** Indicates the table(s) from which data will be obtained.
  - **WHERE:** Indicates the conditions to include a tuple in the result.
  - **GROUP BY:** Indicates the categorization of tuples.
  - **HAVING:** Indicates the conditions to include a category.
  - **ORDER BY:** Sorts the result according to specified criteria.
- The result of an SQL query is a relation.
  - The result table can contain duplicate rows
  - To force the elimination of duplicates, insert the keyword **DISTINCT** after **SELECT.**

SELECT statement examples

SQL Statement	Meaning
SELECT COUNT(*) FROM *T*	Count how many tuples are stored in table *T*
SELECT * FROM *T*	List the content of table *T*
SELECT * FROM *T* LIMIT *n*	Only list *n* tuples from a table
SELECT * FROM *T* ORDER BY *a*	Order the result by attribute *a* (in ascending order; add DESC for descending order)

COUNT() function accepts either ALL, DISTINCT, or *

The SELECT – FROM – WHERE command

- **Example 1:** Which station commence after 1900-1-1?

```
SELECT siteName, commence, orgcode
FROM station
WHERE commence > ‘1900-1-1’;
```

- **Example 2:** How many measurements have we done?

```
SELECT COUNT(*) FROM Measurement;
```

- **Example 3:** List top five measurements ordered by date in descending order.

```
SELECT * FROM Measurement
ORDER BY date DESC limit 5;
```

- SQL is **case-insensitive** and additional spaces and newlines are ignored; use this to format a query for better readability.

SQL data types

- **Integers**
  - Standard integer arithmetic and comparisons available.
- **Floats, Numeric**
  - Floating point numbers with many mathematical operators and functions.
- **Strings (CHAR, VARCHAR)**
  - SQL string literals must be enclosed in single quotes ('like this’).
  - CHAR: fixed length;   VARCHAR: variable length strings up-to max length.
  - String comparison is case-sensitive.
  - Pattern matching with LIKE operator and % placeholders.
  - String concatenation: || (eg. 'hello' || 'there').
- **Date, Timestamp**

Comparison operations in SQL

- Comparison operators in SQL: **= , > , >= , < , <= , != , <>, BETWEEN**.
- Comparison results can be combined using logical connectives: **AND, OR, NOT**.
- **Example 1:**

```
SELECT *
FROM TelescopeConfig
WHERE( mindec BETWEEN -90 AND -50 )
     AND ( maxdec >= -45 )
     AND ( tele_array = 'H168' );
```

- **Example 2:**

```
SELECT *
FROM TelescopeConfig
WHERE tele_array LIKE 'H%';
```

Date and time in SQL

SQL Type	Example	Description
DATE	‘2012-03-26’	A date (some systems incl. time)
TIME	‘16:12:05’	A time, often down to nanoseconds
TIMESTAMP	‘2012-03-26 16:12:05’	Time at a certain date: SQL Server: DATETIME
INTERVAL	‘5 DAY’	A time duration

- Comparisons
  - Normal time-order comparisons with =, >, <, <=, >=, …
- Constants
  - **CURRENT_DATE**: db system’s current date.
  - **CURRENT_TIME**: db system’s current timestamp.
- Example:

```
SELECT *
FROM Epoch
WHERE startDate < CURRENT_DATE;
```

Date and time in SQL (cont'd)

- Database systems support a variety of date/time related operations.
  - Unfortunately, not very standardized – a lot of slight differences.
- Main Operations
  - **EXTRACT**(component **FROM** date).
    - e.g., **EXTRACT**(year **FROM** startDate)
  - **DATE** string (Oracle syntax: **TO_DATE**(string,template))
    - e.g., **DATE** '2012-03-01'
    - Some systems allow templates on how to interpret string.
    - Oracle syntax: **TO_DATE**('01-03-2012', 'DD-Mon-YYYY')
  - **+/- INTERVAL**
    - e.g. '2012-04-01' **+ INTERVAL** '36 HOUR'

JOIN: Querying multiple tables

- Often data that is stored in multiple different relations must be combined.
- We say that the relations are joined.
  - **FROM** clause lists all relations involved in the query.
  - Join-predicates can be explicitly stated in the **WHERE** clause; do not forget it!
- **Examples:**
  - Produces the cross-product Station x Organisation:

```
SELECT *
FROM Station, Organisation;
```

  - Find the site name, commence date and organisation name of all stations:

```
SELECT sitename, commence, organisation
FROM Station, Organisation
WHERE orgcode = code;
```

SQL Join Operators

- SQL offers join operators to directly formulate the natural join, equi-join, and the theta join operations.
  - `R NATURAL JOIN S`
  - `R [INNER] JOIN S ON join-condition`
  - `R [INNER] JOIN S USING (list-of-attributes)`
- These additional operations are typically used in the **FROM** clause.
- **Examples:**
  - List all details of the first three measurements including Water data:

    ```
    SELECT *
    FROM Measurement JOIN Sensor USING (sensor) LIMIT 3;
    ```

  - Find the site name, commence date and organisation name of all stations:

    ```
    SELECT sitename, commence, organisation
    FROM Station JOIN Organisation ON orgcode = code;
    ```

Semantics of a Join

- A Select-From-Where (SFW) query is equivalent to an RA expression
  - **SELECT** `A`$_1$`, A`$_2$`, …, A`$_n$
    FROM `R`$_1$`, R`$_2$`, …, R`$_m$
    WHERE `condition`
- The semantics of a join is as follows
  1. Take **Cartesian product:** R_1 x R_2 x … x R_m
  2. Apply **selection conditions:** `condition`
  3. Apply **projections** to get final output: `A`$_1$`, A`$_2$`, …, A`$_n$

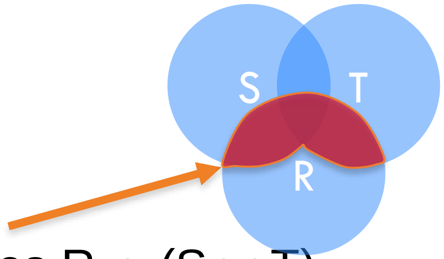
Remark 1: Remembering this order is critical to understanding the output of certain queries (see later on…)

Remark 2: This shows *what a join means, but* not actually how the DBMS executes it

An Unintuitive Query

- Consider three tables R, S, T, each of which contains a single attribute A with integer type, what does the following SQL computes?

  -
    ```
    SELECT DISTINCT R.A
      FROM R, S, T
    WHERE R.A = S.A OR R.A = T.A
    ```

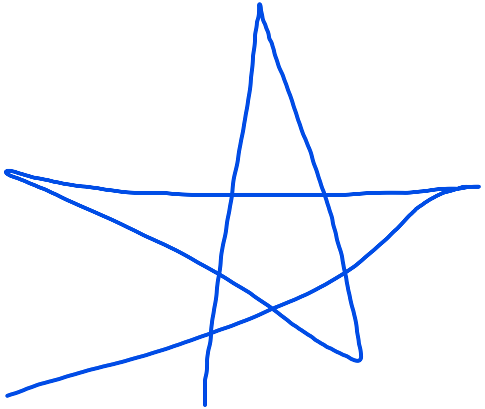


Computes R ∩ (S ∪ T)

But what if S = ϕ?

An Unintuitive Query

```
› SELECT DISTINCT R.A
    FROM R, S, T
  WHERE R.A = S.A OR R.A = T.A
```



– Recall the semantics!

1. Take cross-product First
2. Apply selection conditions
3. Apply projection

– If S = ϕ, then the cross product of R, S, T = ϕ, and the query result = ϕ!

NULL values

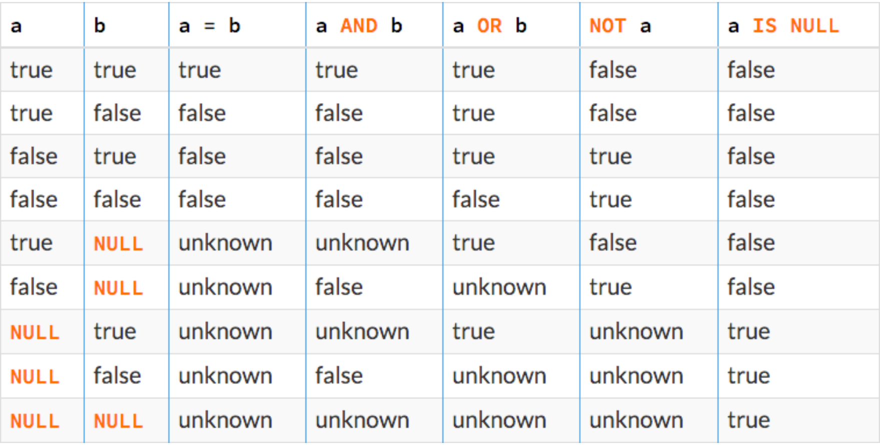
- Tuples can have missing values for some attributes, denoted by **NULL.**
  - Integral part of SQL to handle missing/unknown information.
  - **NULL** signifies that a **value does not exist,** it does not mean “0” or “blank”.
- The predicate **IS NULL** or **IS NOT NULL** can be used to check for nulls.
  - e.g., find measurements with an unknown intensity error value.

```
SELECT gid, band, epoch
FROM Measurement
WHERE intensity IS NULL
```

- Consequence: three-valued logic.
  - The result of any arithmetic expression involving null is null.
    - e.g., 5 + null returns null.
  - However, (most) aggregate functions simply ignore nulls.

NULL values and three-valued logic

- Any comparison with null returns unknown
  - e.g., `5 < null` or `null <> null` or `null = null`



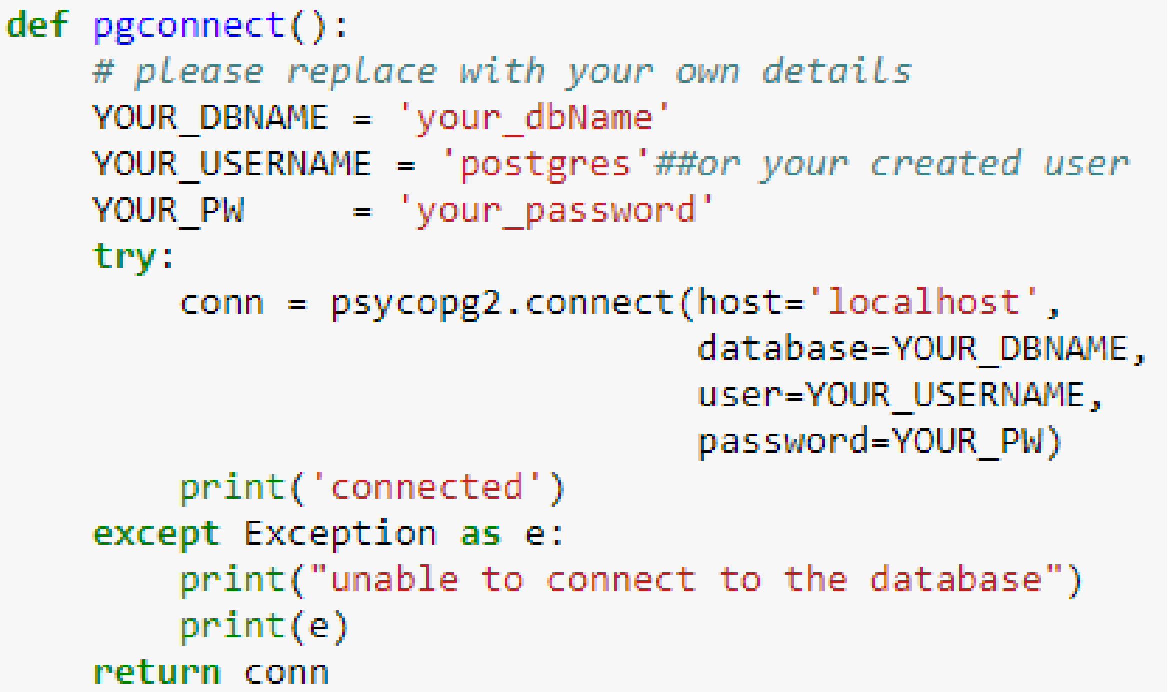
a	b	a = b	a AND b	a OR b	NOT a	a IS NULL
true	true	true	true	true	false	false
true	false	false	false	true	false	false
false	true	false	false	true	true	false
false	false	false	false	false	true	false
true	NULL	unknown	unknown	true	false	false
false	NULL	unknown	false	unknown	true	false
NULL	true	unknown	unknown	true	unknown	true
NULL	false	unknown	false	unknown	unknown	true
NULL	NULL	unknown	unknown	unknown	unknown	true

- Result of **WHERE** clause predicate is treated as false if it evaluates to unknown.
  - e.g., `SELECT sid FROM enrolled WHERE grade = 'unknown'` (ignores all students without a grade so far).

和 null 不同

Reprise: Accessing PostgreSQL from Python: psycopg2

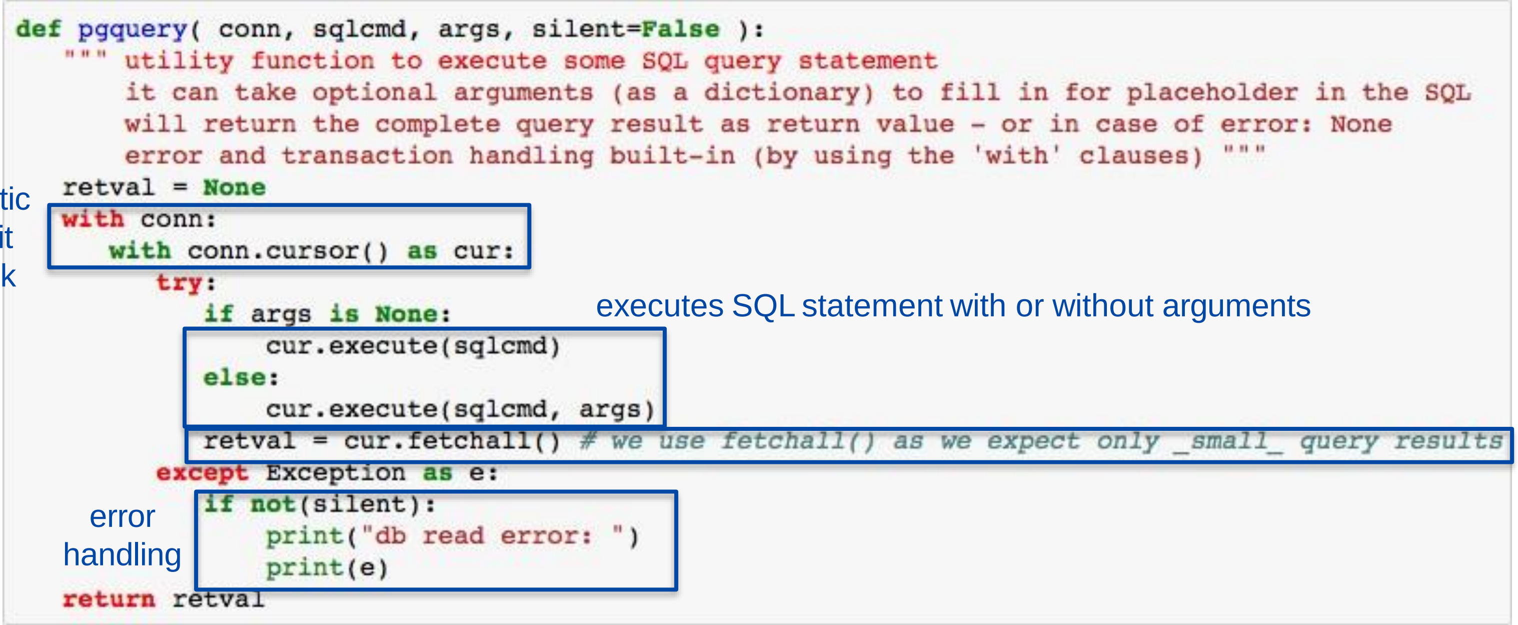
– First, we need to import psycopg2, then connect to PostgreSQL.



```python
def pgconnect():
    # please replace with your own details
    YOUR_DBNAME = 'your_dbName'
    YOUR_USERNAME = 'postgres'##or your created user
    YOUR_PW     = 'your_password'
    try:
        conn = psycopg2.connect(host='localhost',
                                database=YOUR_DBNAME,
                                user=YOUR_USERNAME,
                                password=YOUR_PW)
        print('connected')
    except Exception as e:
        print("unable to connect to the database")
        print(e)
    return conn
```

Querying PostgreSQL from Python

– How to execute an SQL statement on a given connection 'conn’.



```
def pgquery( conn, sqlcmd, args, silent=False ):
    """ utility function to execute some SQL query statement
        it can take optional arguments (as a dictionary) to fill in for placeholder in the SQL
        will return the complete query result as return value - or in case of error: None
        error and transaction handling built-in (by using the 'with' clauses) """
    retval = None
    with conn:
        with conn.cursor() as cur:
            try:
                if args is None:
                    cur.execute(sqlcmd)
                else:
                    cur.execute(sqlcmd, args)
                retval = cur.fetchall() # we use fetchall() as we expect only _small_ query results
            except Exception as e:
                if not(silent):
                    print("db read error: ")
                    print(e)
    return retval
```

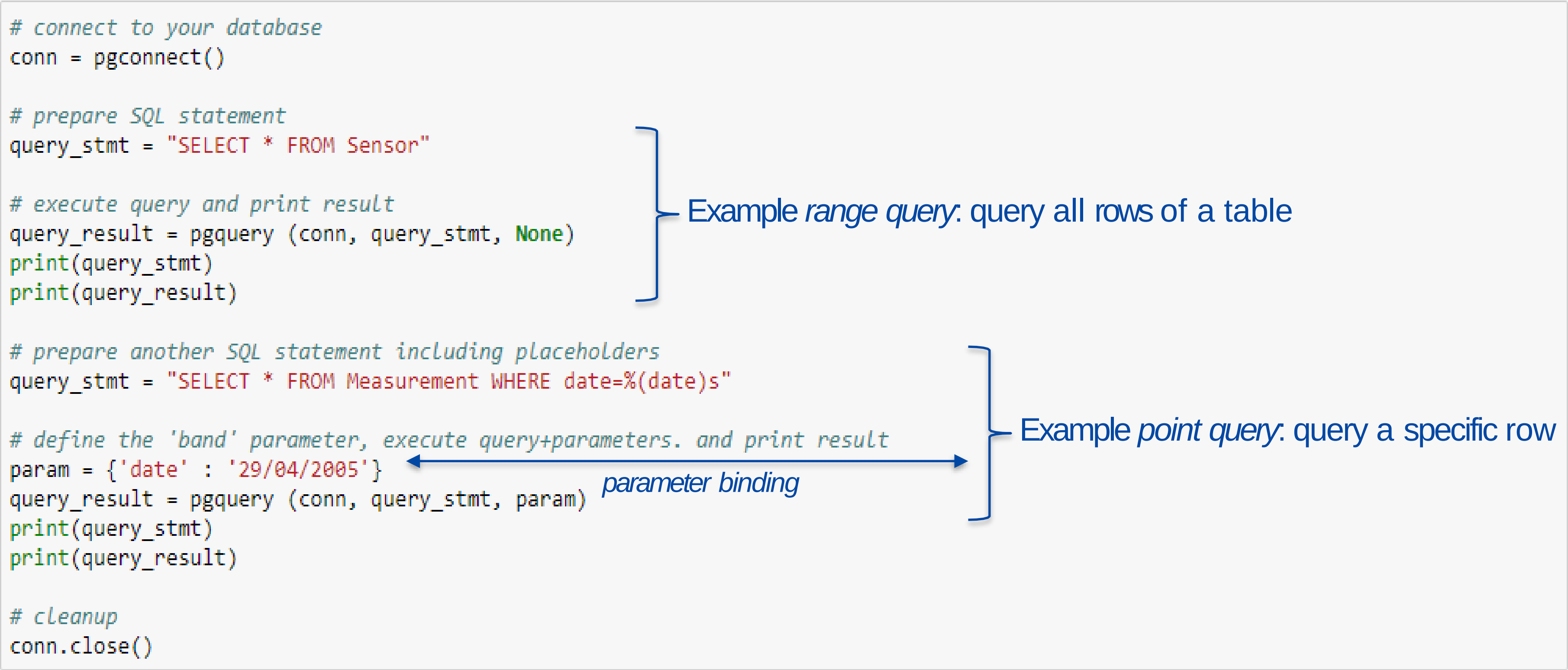
automatic commit rollback

executes SQL statement with or without arguments

error handling

Querying PostgreSQL from Python

– **Example:** Retrieving some data from the database.



```
# connect to your database
conn = pgconnect()

# prepare SQL statement
query_stmt = "SELECT * FROM Sensor"

# execute query and print result
query_result = pgquery (conn, query_stmt, None)
print(query_stmt)
print(query_result)

# prepare another SQL statement including placeholders
query_stmt = "SELECT * FROM Measurement WHERE date=%(date)s"

# define the 'band' parameter, execute query+parameters. and print result
param = {'date' : '29/04/2005'}
query_result = pgquery (conn, query_stmt, param)
print(query_stmt)
print(query_result)

# cleanup
conn.close()
```

Example *range query*: query all rows of a table

Example *point query*: query a specific row

parameter binding

SUMMARISING DATA WITH SQL

THE UNIVERSITY OF SYDNEY

Summarising a database with SQL

- With SQL we can do:
  - Data categorization and aggregation.
  - Complex filtering.
  - Nested queries.
  - Ranking.
  - Etc.
- Basis of data summarisation is the **GROUP BY** clause.

SQL Aggregate Functions

SQL Aggregate Function	Meaning
COUNT(*attr*) ; COUNT(*)	Number of *Not-null-attr* ; or of all values
MIN(*attr*)	Minimum value of *attr*
MAX(*attr*)	Maximum value of *attr*
AVG(*attr*)	Average value of *attr*(arithmetic mean)
MODE() WITHIN GROUP (ORDER BY *attr*)	Mode function over *attr*
PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY *attr*)	Median of the *attr* values

SQL grouping

- So far, we’ve applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several groups of tuples.
- **Example:** Find company and total amount of sales.

Sales Table

company	amount
IBM	5500
DELL	4500
IBM	6500

✗

```
SELECT Company, SUM(Amount)
  FROM Sales
```

company	amount
IBM	16500
DELL	16500
IBM	16500

✓

```
SELECT Company, SUM(Amount)
  FROM Sales
 GROUP BY Company
```

company	amount
IBM	12000
DELL	4500

Queries with GROUP BY and HAVING

- In SQL, we can "partition" a relation into groups according to the value(s) of one or more attributes:

```
        SELECT target-list
        FROM relation-list
        WHERE qualification
        GROUP BY grouping-list
        HAVING group-qualification
```

- A group is a set of tuples that have the same value for all attributes in the grouping-list.
- Attributes in **SELECT** clause outside of aggregate functions must appear in the grouping-list.
  - Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group.

Example: Filtering groups with HAVING clause

- **GROUP BY** example:
  - What was the average mark of each unit of study?

```
SELECT uos_code AS unit_of_study, AVG(mark)
FROM Assessment
GROUP BY uos_code
```

- **HAVING** clause can further filter groups to fulfil a predicate:

```
SELECT uos_code AS unit_of_study, AVG(mark)
FROM Assessment
GROUP BY uos_code
HAVING AVG(mark) > 10
```

- Predicates in the **HAVING** clause are applied after the formation of groups whereas predicates in the **WHERE** clause are applied before forming groups. The **HAVING** clause was added to SQL because the **WHERE** keyword cannot be used with aggregate functions

Evaluation example

– Find the average marks of 6cp unit of studies with more than 2 results.

```
SELECT uos_code AS unit_of_study, AVG(mark)
FROM Assessment NATURAL JOIN UnitOfStudy
WHERE credit_points = 6
GROUP BY uos_code
HAVING COUNT(*) > 2
```

1. Assessment and UnitOfStudy are joined

uos_code	sid	emp_id	mark	title	cpts.	lecturer
COMP5138	1001	10500	60	RDBMS	6	10500
COMP5138	1002	10500	55	RDBMS	6	10500
COMP5138	1003	10500	78	RDBMS	6	10500
COMP5138	1004	10500	93	RDBMS	6	10500
~~ISYS3207~~	~~1002~~	~~10500~~	~~67~~	~~IS Project~~	~~4~~	~~10500~~
~~ISYS3207~~	~~1004~~	~~10505~~	~~80~~	~~IS Project~~	~~4~~	~~10505~~
SOFT3000	1001	10505	56	C Prog.	6	10505
~~INFO2120~~	~~1005~~	~~10500~~	~~63~~	~~DBS 1~~	~~4~~	~~10500~~
…	…	…	….	…	…	…

2. Tuples that fail the **WHERE** condition are discarded

Evaluation example

3. remaining tuples are partitioned into groups
   by the value of attributes in the grouping-list.

uos_code	sid	emp_id	mark	title	cpts.	lecturer
COMP5138	1001	10500	60	RDBMS	6	10500
COMP5138	1002	10500	55	RDBMS	6	10500
COMP5138	1003	10500	78	RDBMS	6	10500
COMP5138	1004	10500	93	RDBMS	6	10500
~~SOFT3000~~	~~1001~~	~~10505~~	~~56~~	~~C Prog.~~	~~6~~	~~10505~~
INFO5990	1001	10505	67	IT Practice	6	10505
…	…	…	….	…	…	…

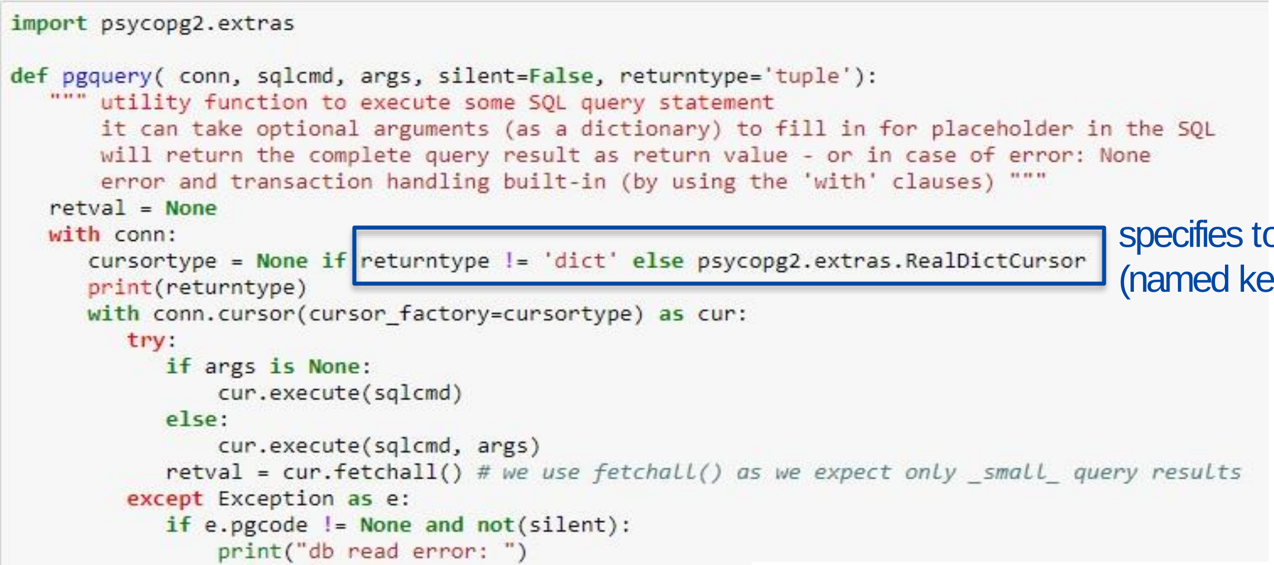
4. Groups which fail
   the **HAVING** condition
   are discarded.

5. Each group will get ONE answer tuple

uos_code	AVG(..)
COMP5138	56
INFO5990	40.5

GATHERING DATA FOR VISUALIZATION

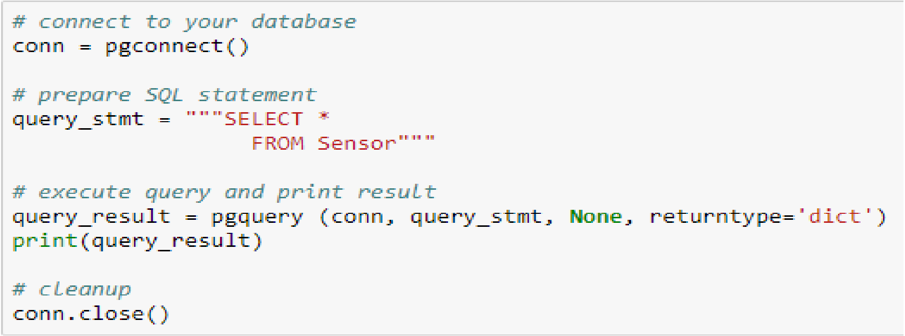
Data gathering for visualisation from SQL in Python



```
import psycopg2.extras

def pgquery( conn, sqlcmd, args, silent=False, returntype='tuple'):
   """ utility function to execute some SQL query statement
       it can take optional arguments (as a dictionary) to fill in for placeholder in the SQL
       will return the complete query result as return value - or in case of error: None
       error and transaction handling built-in (by using the 'with' clauses) """
   retval = None
   with conn:
      cursortype = None if returntype != 'dict' else psycopg2.extras.RealDictCursor
      print(returntype)
      with conn.cursor(cursor_factory=cursortype) as cur:
         try:
            if args is None:
                cur.execute(sqlcmd)
            else:
                cur.execute(sqlcmd, args)
            retval = cur.fetchall() # we use fetchall() as we expect only _small_ query results
         except Exception as e:
            if e.pgcode != None and not(silent):
                print("db read error: ")
                print(e)
   return retval
```

specifies to return each result row as a dictionary (named key-value pairs)



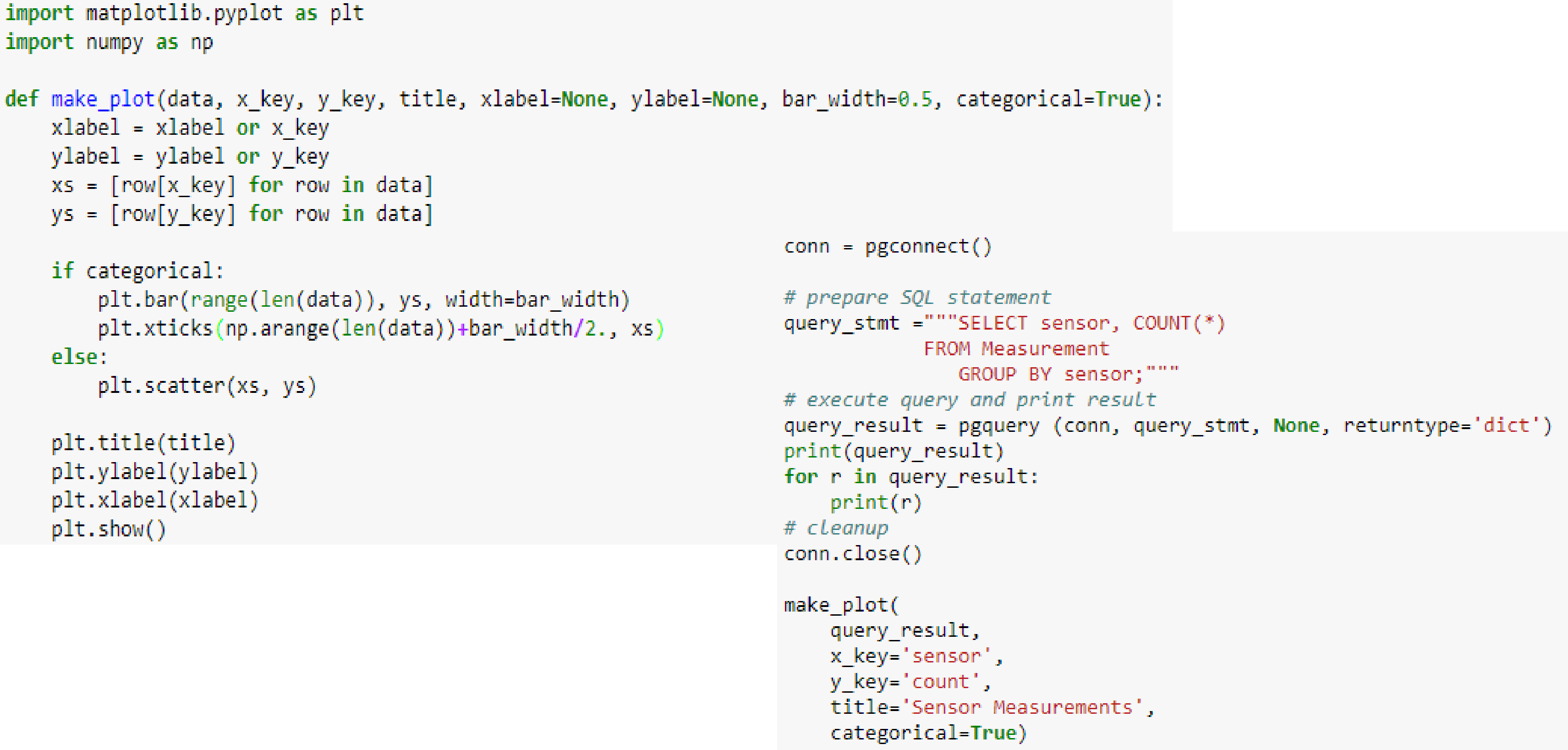
```
# connect to your database
conn = pgconnect()

# prepare SQL statement
query_stmt = """SELECT *
                FROM Sensor"""

# execute query and print result
query_result = pgquery (conn, query_stmt, None, returntype='dict')
print(query_result)

# cleanup
conn.close()
```

Data visualisation from SQL in Python



```
import matplotlib.pyplot as plt
import numpy as np

def make_plot(data, x_key, y_key, title, xlabel=None, ylabel=None, bar_width=0.5, categorical=True):
    xlabel = xlabel or x_key
    ylabel = ylabel or y_key
    xs = [row[x_key] for row in data]
    ys = [row[y_key] for row in data]

    if categorical:
        plt.bar(range(len(data)), ys, width=bar_width)
        plt.xticks(np.arange(len(data))+bar_width/2., xs)
    else:
        plt.scatter(xs, ys)

    plt.title(title)
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)
    plt.show()
```

```
conn = pgconnect()

# prepare SQL statement
query_stmt ="""SELECT sensor, COUNT(*)
            FROM Measurement
               GROUP BY sensor;"""
# execute query and print result
query_result = pgquery (conn, query_stmt, None, returntype='dict')
print(query_result)
for r in query_result:
    print(r)
# cleanup
conn.close()

make_plot(
    query_result,
    x_key='sensor',
    y_key='count',
    title='Sensor Measurements',
    categorical=True)
```

Review

W5 review: Querying and summarising data

Objective

- To be able to extract a data set from a database, as well as to leverage on the SQL capabilities for in-database data summarisation and analysis.

Lecture

- Data gathering reprise.
- SQL querying.
- Summarising data with SQL.
- Statistic functions support in SQL.

Readings

- Data Science from Scratch: Ch 24.

Exercises

- Data Loading.
- SQL Querying.
- Python DB Querying.
- Data Summarization using SQL.

TO-DO in W5

- Finish Ed Lessons Python modules.
- Finish Ed Lessons SQL modules.