# INFO3406
# Introduction to Data Analytics

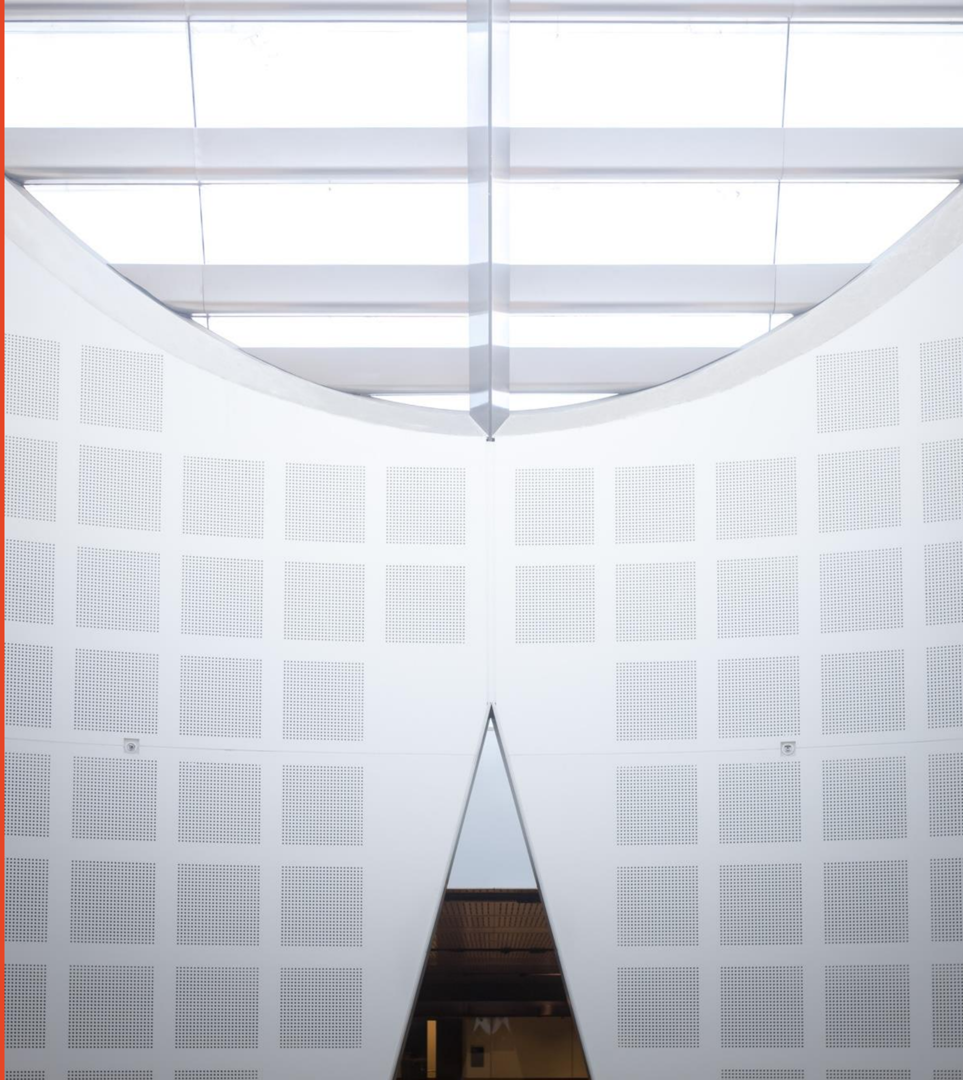
## W4: Data Extract, Data Transformation and Storage

**Presented by**

Dr Ali Anaissi

School of IT

THE UNIVERSITY OF SYDNEY



# Overview of Week 4

THE UNIVERSITY OF SYDNEY

# Today: 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 9 + 10

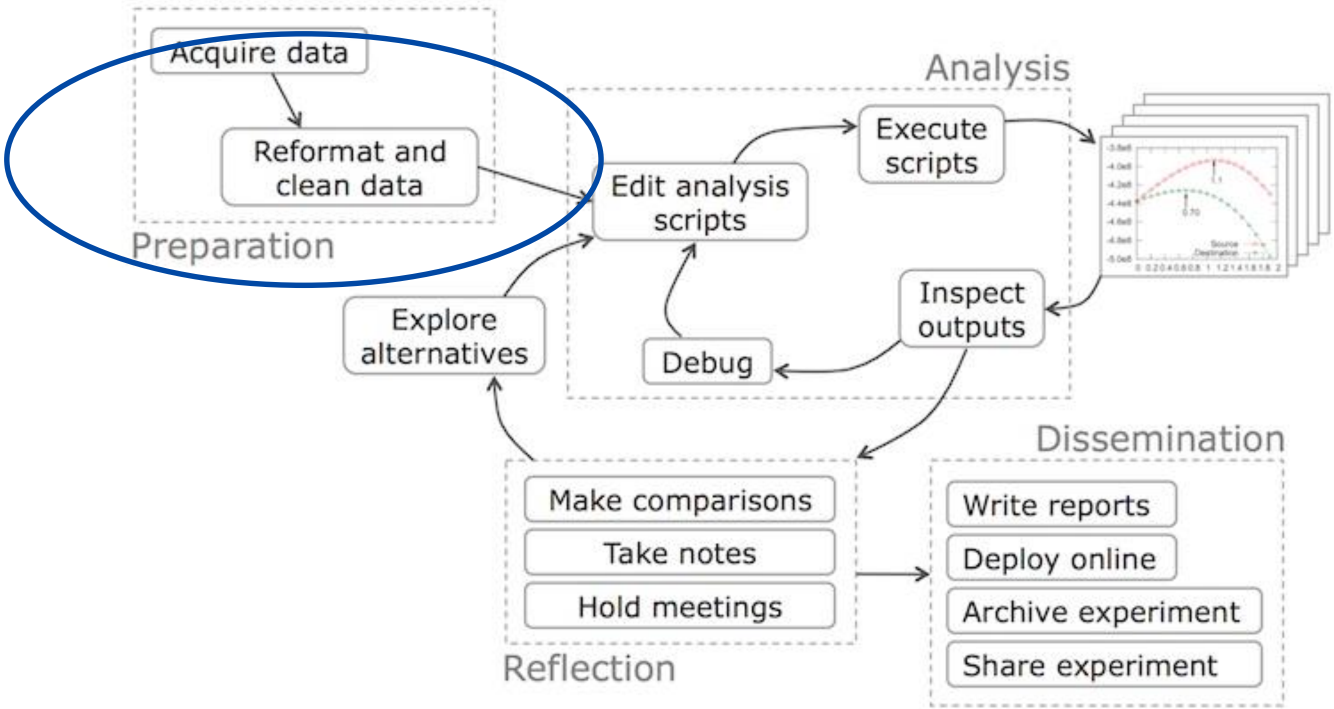
**Exercises**

- Python / Jupyter to load data
- psycopg2
- PostgreSQL to store data

**TODO in W4**

- Grok Python modules
- Grok SQL modules
- Summarise and prepare data

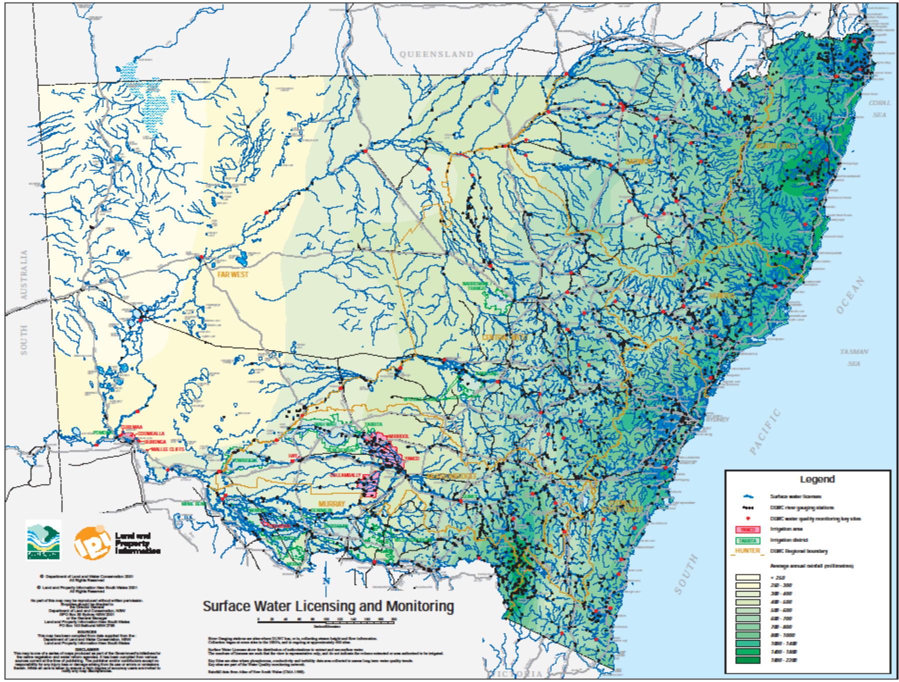
# Exploratory Analysis Workflow



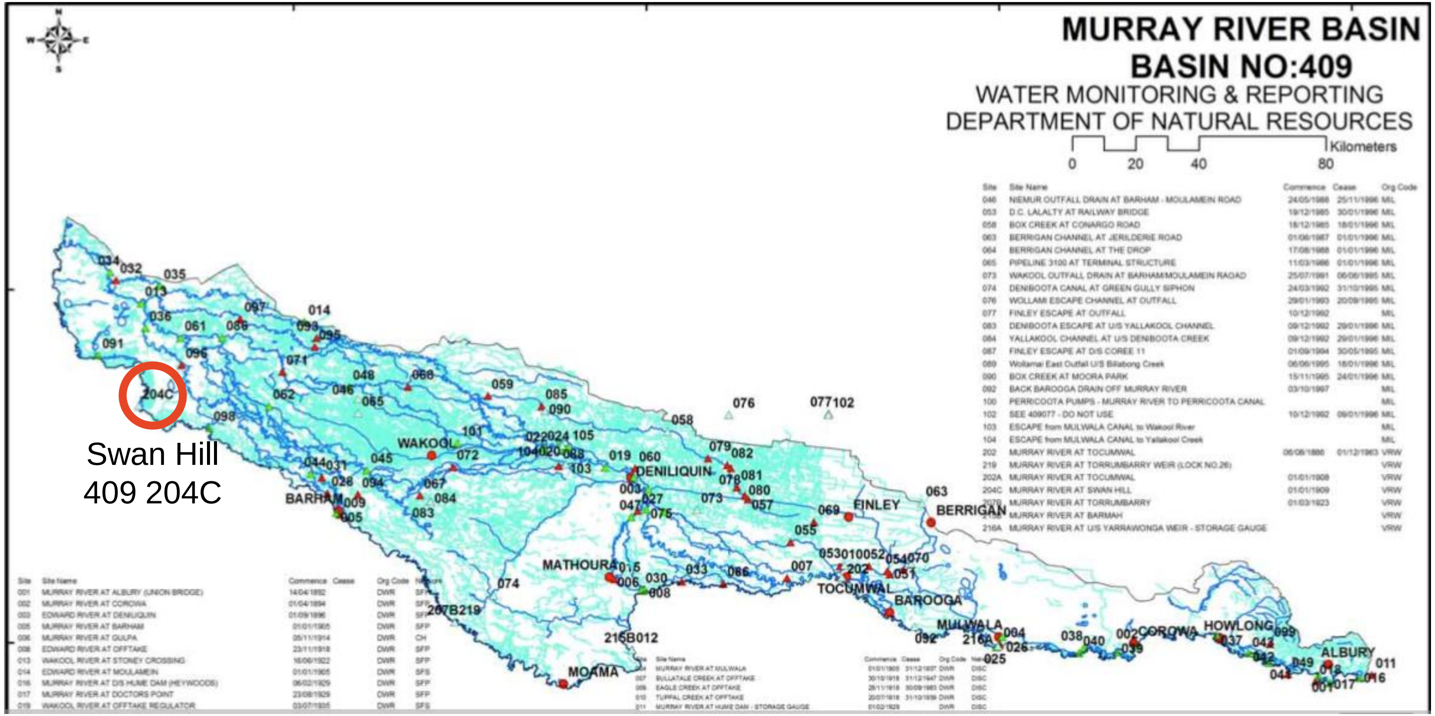
# New Scenario

# New Data Set

- Water measurements:
  - automatic monitoring stations that are distributed over a larger area
  - Periodically send their measured values to a central authority
  - Time-series data of:
    - water level
    - water flow
    - water temperature
    - salinity (via measuring electric conductivity) or other hydraulic properties



# Example: Murray River Basin in NSW




[Source: www.waterinfo.nsw.gov.au]

# Where do we get data from?

- You or your organization might have it already, or a colleagues provides you access to data.
  - Typical exchange formats:  CSV, Excel, XML/JSON
- Or: Download from an online data server
  - Still typically in CSV or Excel etc, but now problems with meta-data
- Or: Scrap the web yourself or use APIs of resources
  - Cf. textbook, chapter 9

- Our data set comes from a colleague in Excel format

# Water dataset

Contains five CSV data files:

- Measurements.csv
- Organisations.csv
- Sensors.csv
- Stations.csv

- Lets have  a look

# Relational Databases

- Today's goal is to store the data in a relational database

- **Relational data model** is the most widely used model today
  - Main concept: **relation**, basically a table with rows and columns
  - Every relation has a **schema**, which describes the columns, or fields

- This sounds like a spreadsheet, but as we will see, it has some differences

# Definition of Relation

- **Informal Definition:**
  A ***relation*** is a named, two-dimensional table of data
  - Table consists of rows (record) and columns (attribute or field)
- Example:

Attributes (also: columns, fields)

| ***Student*** | | | | |
|---|---|---|---|---|
| <u>sid</u> | name | login | gender | address |
| 5312666 | Jones | ajon1121@cs | m | 123 Main St |
| 5366668 | Smith | smith@mail | m | 45 George |
| 5309650 | Jin | ojin4536@it | f | 19 City Rd |

Tuples (rows, records)

# Some Remarks

- Not all tables qualify as a relation:
  - Every relation must have a unique name.
  - Attributes (columns) in tables must have unique names.
    => The order of the columns is irrelevant.
  - All tuples in a relation have the same structure;
    constructed from the same set of attributes
  - Every attribute value is atomic (not multivalued, not composite).
  - Every row is unique
    (can’t have two rows with exactly the same values for all their fields)
  - The order of the rows is immaterial

# First attempt for database storage

- Our first attempt is for just a 1:1 mapping
  - Still: separate tables need to go to separate database relations
  - Note: CSV headers are not allowed to contain spaces or '
- Connection to Postgresql with **psql** shell tool
- Create 1:1 mapping tables in SQL
- Load CSV directly to SQL tables
  - We first try COPY command from psql
    http://www.postgresql.org/docs/current/interactive/sql-copy.html

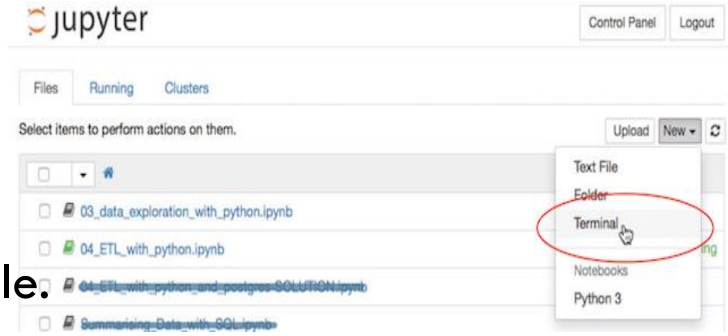
# DB Creation and Data Import using pgsql

- Here we can work with a postgresql database using the 'psql' command.

  psql -h soit-db-pro-2.ucc.usyd.edu.au -U y18s2i3406_< your_unikey >

- We give the following SQL create table statement:

```
DROP TABLE IF EXISTS Organisation;
CREATE TABLE IF NOT EXISTS Organisation (
    code VARCHAR(20) PRIMARY KEY,
    organisation    VARCHAR(150));
```




- Next we want to load data from an external CSV file.
- We will use psql's **copy** command for this.

  \copy Organisation (code,organisation) FROM 'Organisations.csv' WITH CSV HEADER

- Psql's \copy command is quite useful -- as long as table and CSV files directly match, and as long as the CSV file's content is in good shape. Otherwise it soon reaches its limits.

# Issues with DB Loaders

- DB Loading tools such as **psql**
  - good for administration of server
  - good for bulk-loading of exported data in clean csv format (db export)
  - needs terminal access (both an advantage and a disadvantage)
  - has its limitations if format and structure of data files do not match the actual database schema
    => cleaning and transformation of data needed
- Could we do so in a programming language such as Python?

# Database Loading with Python

# Accessing PostgreSQL from Python: psycopg2

- First, we need to import the **psycopg2** module, then connect to Postgresql
- Note: You need obviously to provide you own login name
  - Username and password are prepared by us as your unikey and your SID

```
import psycopg2

def pgconnect():
    # please replace <your_unikey> and <your_SID> with your own details
    YOUR_UNIKEY = '<your_unikey>'
    YOUR_PW     = '<your_SID>'
    try:
        conn = psycopg2.connect(host='soit-db-pro-2.ucc.usyd.edu.au',
                                database='y18s2i3406_'+YOUR_UNIKEY,
                                user='y18s2i3406_'+YOUR_UNIKEY,
                                password=YOUR_PW)
        print('connected')
    except Exception as e:
        print("unable to connect to the database")
        print(e)
    return conn
```

# Accessing PostgreSQL from Python: psycopg2 (cont'd)

- How to execute an SQL statement on an open connection 'conn'
  - we prepared a helper function which encapsulates all the error handling:

```
def pgexec( conn, sqlcmd, args, msg ):
    """ utility function to execute some SQL statement
        can take optional arguments to fill in (dictionary)
        error and transaction handling built-in """
    retval = False
    with conn:
        with conn.cursor() as cur:
            try:
                if args is None:
                    cur.execute(sqlcmd)
                else:
                    cur.execute(sqlcmd, args)
                print("success: " + msg)
                retval = True
            except Exception as e:
                print("db error: ")
                print(e)
    return retval
```

# Accessing PostgreSQL from Python: psycopg2 (cont'd)

– Example: Creating a table and loading some data

```
data_organisations = list(csv.DictReader(open('water_data/Organisations.csv')))
# 1st: login to database
conn = pgconnect()

# 2nd: ensure that the schema is in place
organisation_schema = """CREATE TABLE IF NOT EXISTS Organisation (
                         code VARCHAR(20) PRIMARY KEY,
                         orgName    VARCHAR(150)
                      )"""
pgexec (conn, organisation_schema, None, "Create Table Organisation")

# 3nd: load data
# IMPORTANT: make sure the header line of CSV is without spaces!
insert_stmt = """INSERT INTO Organisation(code,orgName)
                   VALUES (%(Code)s, %(Organisation)s)"""
for row in data_organisations:
    pgexec (conn, insert_stmt, row, "row inserted")
```

# Issues could be encountered

- Interpretation of data format and meta-data
- Differences in naming conventions
  - Excel headers with spaces and quotes, which both are not allowed to DBMS
- Inconsistent or missing data entries
- 'shape' of data

# Transforming and Cleaning Data

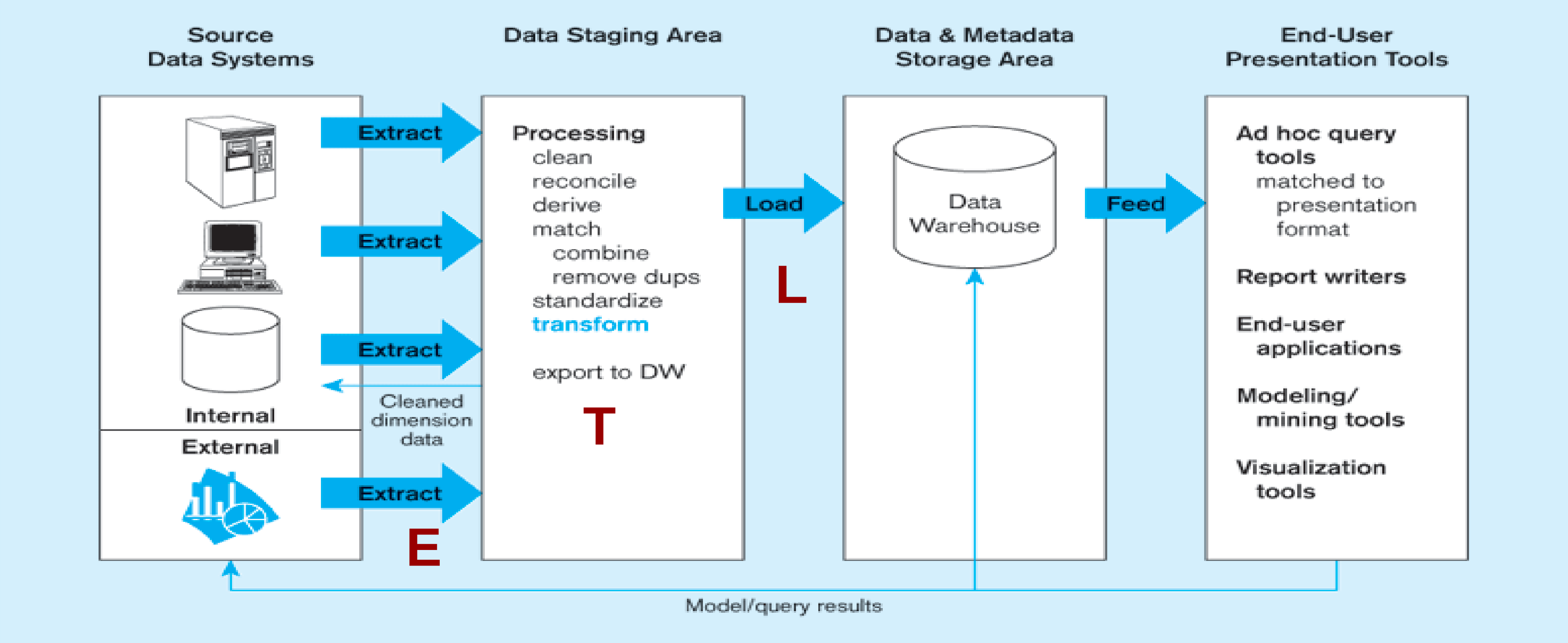
THE UNIVERSITY OF SYDNEY

# Cleaning and Transforming Data

- Real data is often '*dirty*'
- Important to do some data cleaning and transforming first
  - remember last week, when we had to convert strings to float
- Typical steps involved:
  - type and name conversion
  - filtering of missing or inconsistent data
  - unifying semantic data representations
  - matching of entries from different sources
- Later also:
  - **<u>Rescaling</u>** and optional dimensionality reduction

# ETL Process

- This problem is well known from data warehousing
- **ETL Process:** Capture/Extract - Data Cleansing - Transform - Load



# Data Cleaning in Python

- Yes, there are powerful ETL tools out there, but we do it for free in Python:
  (1) type and name conversion
  (2) filtering of missing or inconsistent data
  (3) unifying semantic data representations
  (4) matching of entries from different sources
- Last week's clean() function deals with Tasks (1) and (2)
  - int() creates integer objects, e.g., -1, 101
  - float() creates floating point object, e.g., 3.14, 2.71
  - datetime.strptime() creates datetime objects from strings
  - Filters missing / wrongly formatted data and replaces with default value
- For more complex cases (3) and (4), you would need special code though

# Repeat from Week 3: A function to convert values

**Use “not a number” as default value**
**numpy knows to ignore for some stats**

```python
import numpy as np
DEFAULT_VALUE = np.nan

def clean(data, column_key, convert_function, default_value):
    special_values= {} # no special values yet
    for row in data:
        old_value = row[column_key]
        new_value = default_value
        try:
            if old_value in special_values.keys():
                new_value = special_values[old_value]
            else:
                new_value = convert_function(old_value)
        except (ValueError, TypeError):
            print('Replacing {} with {} in column {}'.format(row[column_key], new_value, column_key))
        row[column_key] = new_value

# the following converts the two measurement columns to float values - or NaN
clean(data_measurements, 'Discharge', float, DEFAULT_VALUE)
clean(data_measurements, 'MeanDischarge', float, DEFAULT_VALUE)
```

# Data Modeling

THE UNIVERSITY OF SYDNEY

# Relation Database Theory and Issues

- The modelling process in relational database known as OLTP (Online Transactional Processing) focuses on normalization process which yields to a flexible model
  - making it easy to maintain dynamic relationships between business entities
- So it is effective and efficient for operational databases – a lot of updates
- However, a fully normalized data model can perform very inefficiently for queries.
- Historical data are usually large with static relationships:
  - Unnecessary joins may take unacceptably long time
- So how to proceed with a database approach?

  => **OLAP: Online Analytical Processing (Data Warehousing Approach)**

# What is a Data Warehouse?

- "A data warehouse is simply a single, complete, and consistent store of data obtained from a variety of sources and made available to end users in a way they can understand and use it in a business context." -- Barry Devlin, *IBM Consultant*

- A data warehouse is a subject-oriented, integrated, time-variant, and nonvolatile collection of data that is used primarily in organizational decision making. --W. H. Inmon
  - OLTP is trying to run a business, while;
  - OLAP is trying to improve/optimize a business
    - » is an element of decision support systems (DSS)

# What is a Data Warehouse?

- Subject-oriented
  - Organized by subject, not by application
  - Used for analysis, data mining, etc.
- Integrated
  - Constructed by integrating multiple, heterogeneous data sources
    - relational databases, flat files, on-line transaction record
- Time Variant
  - Large volume of historical data (Gb, Tb)
  - Time attributes are important
- Non-volatile
  - Updates infrequent or does not occur
  - May be append-only

# Conceptual Modeling of Data Warehouses

- Modeling data warehouses: dimensions & measures instead of relational model
- Data warehouse contains a **large** central table (**fact table**)
  - Contains the data without redundancy
- A set of **dimension tables**

# Data Warehouses: Fact Tables

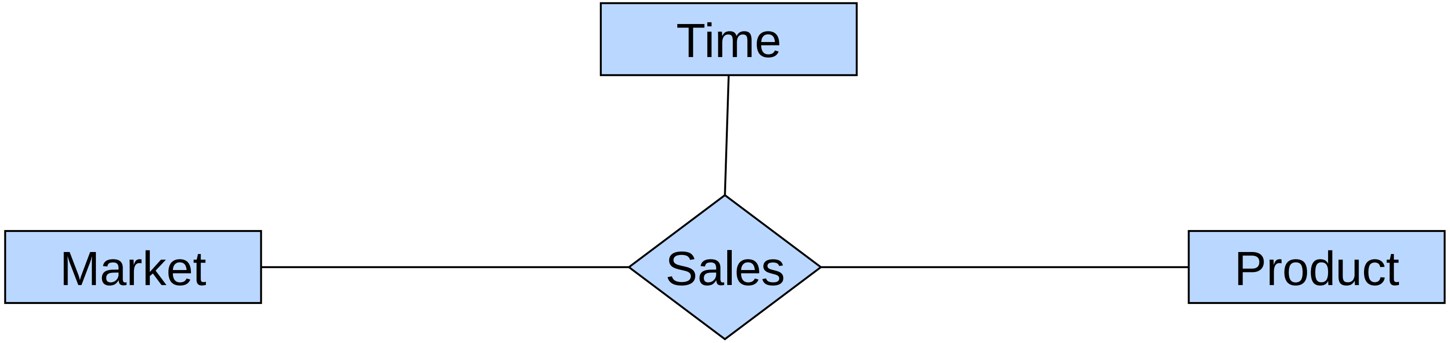
- Relational ‘data warehouse‘ applications are centered around a ***fact table***
  - For example, a supermarket application might be based on a table
    Sales (*Market_Id*, *Product_Id*, *Time_Id*, *Sales_Amt*)

| market_id | product_id | time_id | sales_amt |
|---|---|---|---|
| M1 | P1 | T1 | 3000 |
| M1 | P2 | T1 | 1000 |
| M1 | P3 | T1 | 500 |
| M2 | P1 | T1 | 100 |
| M2 | P2 | T1 | 1100 |
| M2 | P3 | … | … |
| … | … | | |

- The table can be viewed as *multidimensional*
  - Collection of numeric measures, which depend on a set of dimensions
    - E.g. *Market_Id*, *Product_Id*, *Time_Id* are the dimensions that represent specific supermarkets, products, and time intervals
    - *Sales_Amt* is a function of the other three

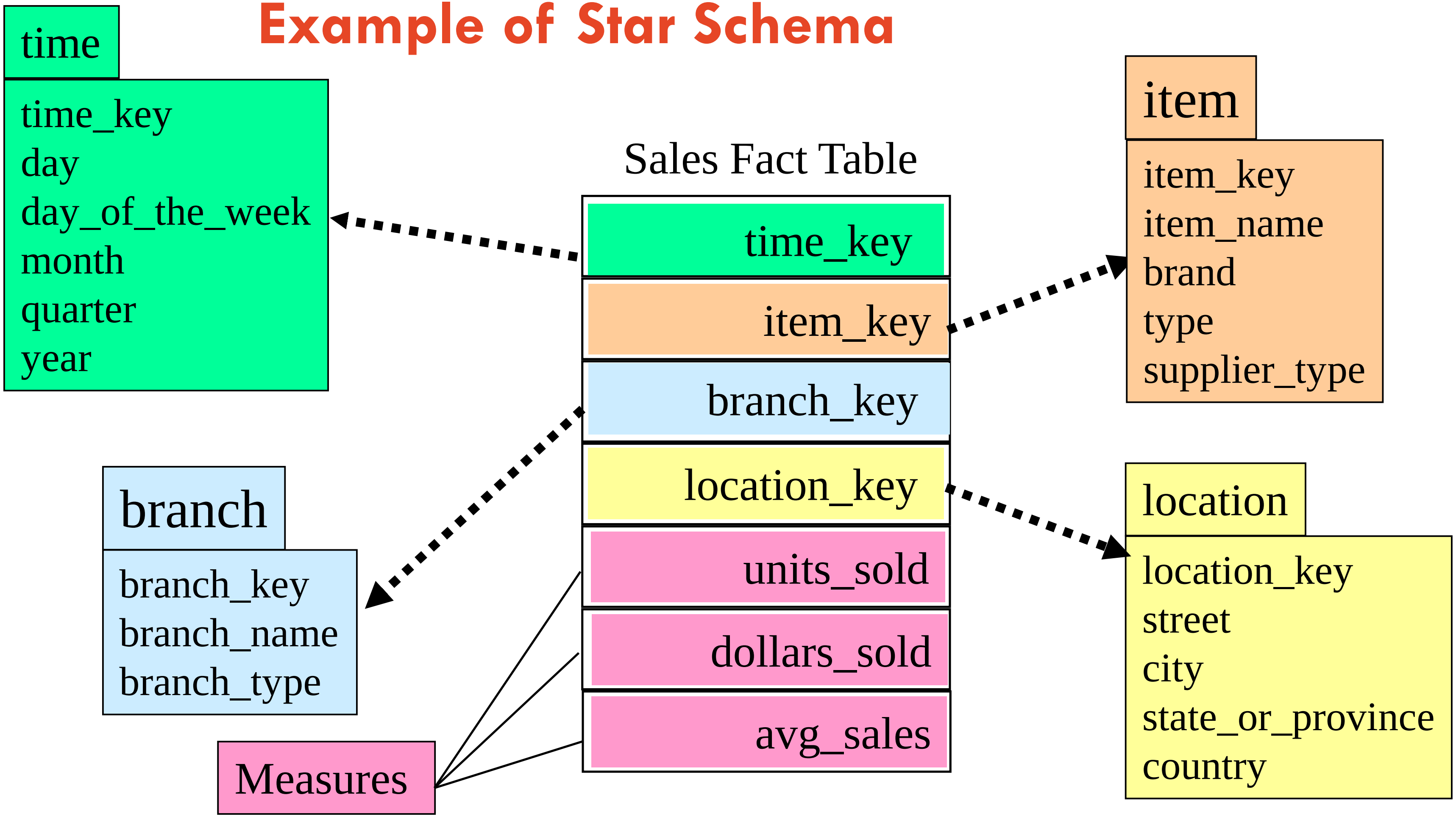
# Data Warehousing: Star Schema

- The fact and dimension relations linked to it looks like a star;
- this is called a ***star schema***
- *Most common modeling paradigm*



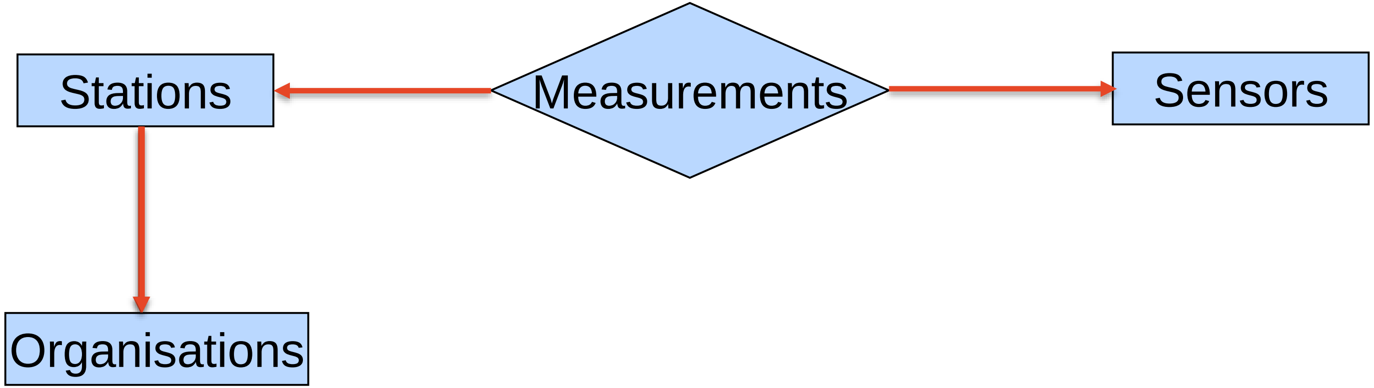

- If we map this to relations
  - 1 central fact table
  - *n* dimension tables with foreign key relationships from the fact table *(the fact table holds the FKs referencing the dimension tables)*

# Example of Star Schema

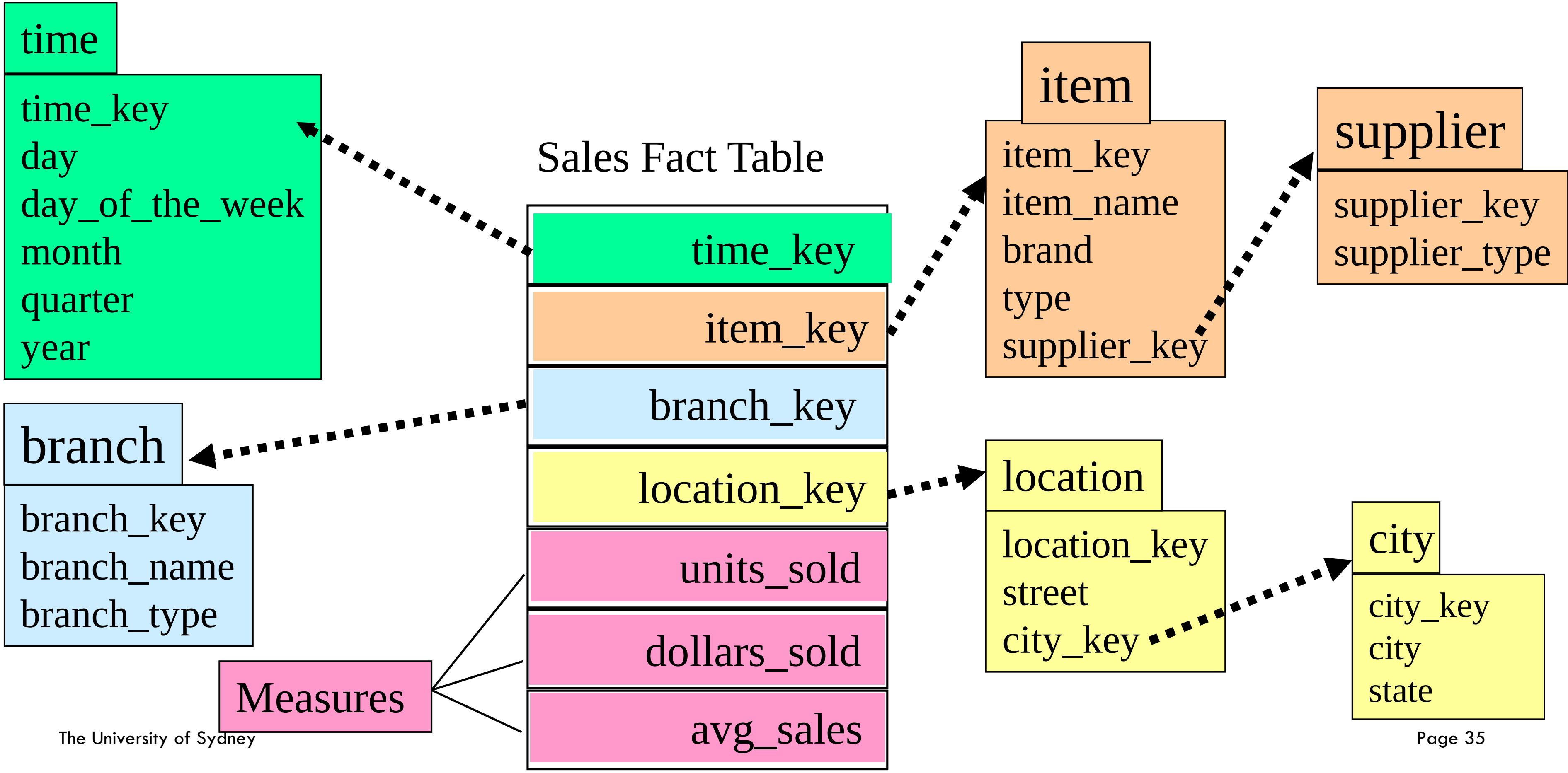


# Data Warehousing: Snowflake Schema

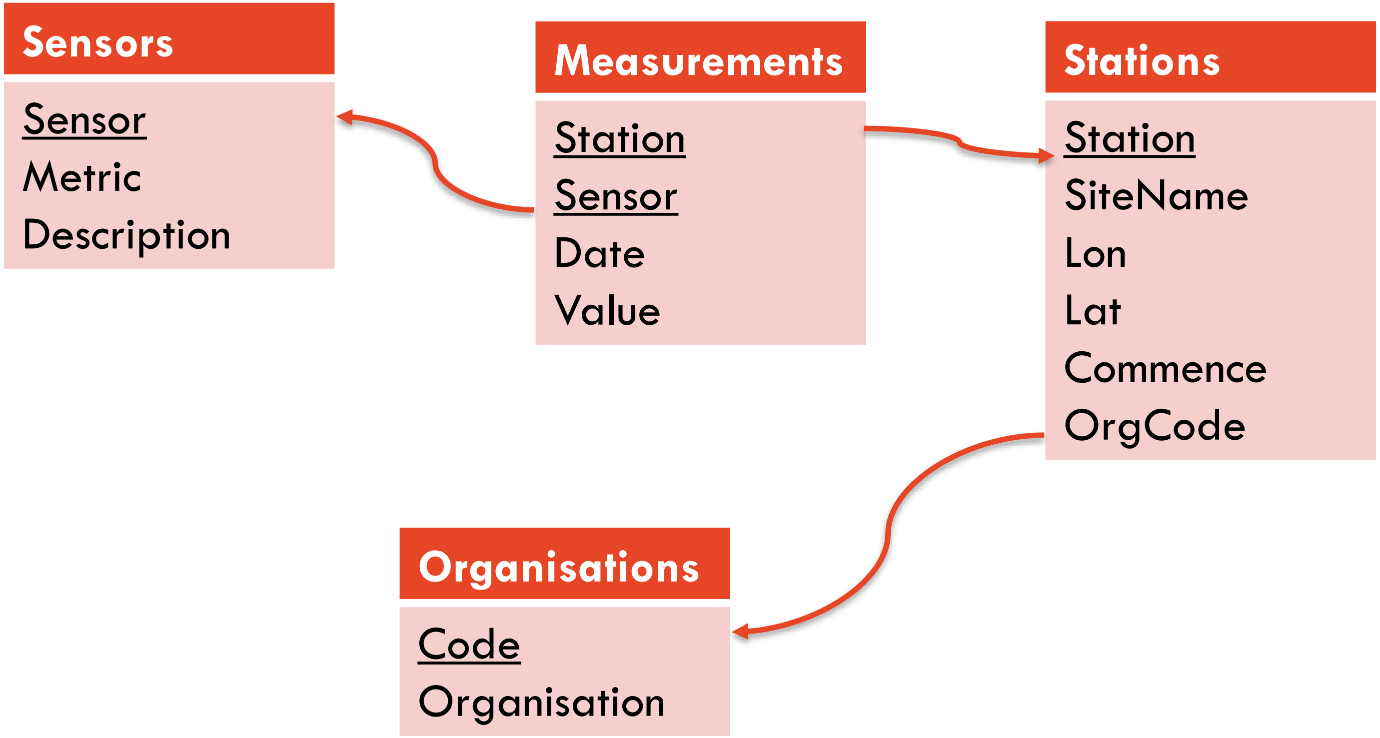
- Snowflake schema:  A refinement of star schema where some dimensional hierarchy is **normalized** into a set of smaller dimension tables, forming a shape similar to snowflake
- measurements are the facts, rest describes the dimensions



# Example of Snowflake Schema



# Modeling our Water Data Set



# Data Warehousing: Fact constellations

– Fact constellations: **Multiple** fact tables share dimension tables, viewed as a collection of stars, therefore called galaxy schema or fact constellation

# DB Creation

# SQL – The Structured Query Language

- SQL is the standard declarative query language for RDBMS
- Supported commands from roughly two categories:
  - **DDL** (Data Definition Language)
    - Create, drop, or alter the relation schema
    - Example:

      **CREATE TABLE** ***name*** **(** ***list_of_columns*** **)**
  - **DML** (Data Manipulation Language)
    - for <u>retrieval</u> of information also called **query language**
    - **INSERT**, **DELETE**, **UPDATE**
    - **SELECT** … **FROM** … **WHERE**

# Table Constraints and Relational Keys

- When creating a table, we can also specify ***Integrity Constraints*** for columns
  - eg. domain types per attribute, or **NULL** / **NOT NULL** constraints

- **Primary key:** unique, minimal identifier of a relation.
  - Examples include employee numbers, social security numbers, etc. This is how we can guarantee that all rows are unique.

- **Foreign keys** are identifiers that enable a dependent relation (on the many side of a relationship) to refer to its parent relation (on the one side of the relationship)
  - Must refer to a candidate key of the parent relation
  - Like a `logical pointer'

- Keys can be **simple** (single attribute) or **composite** (multiple attributes)

# Example: Relational Keys

**Primary key** identifies each tuple of a relation.

**Composite Primary Key** consisting of more than one attribute.

| Student | |
|---|---|
| sid | name |
| 31013 | John |

| Enroll | | |
|---|---|---|
| sid | ucode | grade |
| 31013 | I2120 | CR |

| Units_of_study | | |
|---|---|---|
| ucode | title | credit_pts |
| I2120 | DB Intro | 4 |

**Foreign key** is a (set of) attribute(s) in one relation that \`refers' to a tuple in another relation (like a \`logical pointer').

# SQL Domain Constraints

- SQL supports various domain constraints to restrict attribute to valid domains
  - NULL / NOT NULL — whether an attribute is allowed to become *NULL* (unknown)
  - DEFAULT — to specify a default value
  - CHECK( *condition* ) — a Boolean *condition* that must hold for every tuple in the db instance

Example:

```
CREATE TABLE Student
 (
     sid          INTEGER         PRIMARY KEY,
     name         VARCHAR(20)     NOT NULL,
     gender       CHAR            CHECK (gender IN ('M,'F','T')),
     birthday     DATE            NULL,
     country      VARCHAR(20),
     level        INTEGER         DEFAULT 1 CHECK (level BETWEEN 1 and 5)
 );
```

# Data Loading / Storage

# Data Storing

- Where are we now?
  - We have analysed our given data set
  - Cleaned it
  - Transformed it and created a corresponding relational database
- Next, we want to store the given data in our database.
- Main approaches:
  1. Command line tools
  2. Python loader
  3. (Combination of Python loader and stored procedures)

# Approach 1: PSQL Data Loader

- Postgresql offers a command to load data directly from a CSV file into a database table

  \COPY *tablename* FROM *filename* CSV [HEADER] [NULL '…']

  - Many further options
  - Try \help COPY
- Pros:
  - Relatively fast and straight-forward
  - No programming needed
- Cons:
  - Only 1:1 mapping of CSV to tables; no data cleaning or transformation
  - Stops at the first error…

# Approach 2: Python Loading Code

- Example: Creating a table and loading some data

```python
# 1st: login to database
conn = pgconnect()

# 2nd: ensure that the schema is in place
organisation_schema = """CREATE TABLE IF NOT EXISTS Organisation (
                         code VARCHAR(20) PRIMARY KEY,
                         orgName     VARCHAR(150)
                   )"""
pgexec (conn, organisation_schema, None, "Create Table Organisation")

# 3nd: load data
# IMPORTANT: make sure the header line of CSV is without spaces!
insert_stmt = """INSERT INTO Organisation(code,orgName)
                      VALUES (%(Code)s, %(Organisation)s)"""
for row in data_organisations:
    pgexec (conn, insert_stmt, row, "row inserted")
```

- Pros: Full flexibility; data cleaning and transformation possible
- Cons: Has to be hand-coded for each case

# SQL DML Statements

- Insertion of new data into a table / relation
  - **Syntax:**
    INSERT INTO *table* [“(”*list-of-columns*“)”] VALUES “(“ list-of-*expression* “)”
  - Example:
    **INSERT INTO Students (sid, name) VALUES (53688, ‘Smith’)**
- Updating of tuples in a table / relation
  - **Syntax:**
    UPDATE *table* SET *column*“=“*expression* {“,”*column*“=“*expression*}
    [ WHERE *search_condition* ]
  - Example:
    ```
    UPDATE students
        SET gpa = gpa - 0.1
      WHERE gpa >= 3.3
    ```
- Deleting of tuples from a table / relation
  - **Syntax:**
    DELETE FROM *table* [ WHERE *search_condition* ]
  - Example:
    **DELETE FROM Students WHERE name = ‘Smith’**

# Database Creation

– The full SQL schema for the given data model

```
DROP TABLE IF EXISTS Organisation CASCADE;
CREATE TABLE IF NOT EXISTS Organisation (
   code VARCHAR(20) PRIMARY KEY,
   organisation    VARCHAR(150)
);

DROP TABLE IF EXISTS Station CASCADE;
CREATE TABLE IF NOT EXISTS Station (
     station   VARCHAR(50) PRIMARY KEY,
     siteName  VARCHAR(50),
     commence Date,
     orgCode  VARCHAR(50),
     CONSTRAINT orgCodeFK
          FOREIGN KEY (orgCode)
          REFERENCES Organisation (code)
 );
```

```
DROP TABLE IF EXISTS Sensor CASCADE;
CREATE TABLE IF NOT EXISTS Sensor (
     sensor   VARCHAR(20) PRIMARY KEY,
     description  VARCHAR(150) ,
     metric VARCHAR(20)
  );

DROP TABLE IF EXISTS Measurement CASCADE;
CREATE TABLE IF NOT EXISTS Measurement (
     station   VARCHAR(20),
     sensor   VARCHAR(20),
     date DATE,
     value  FLOAT,
     CONSTRAINT stationFK
          FOREIGN KEY (station)
          REFERENCES Station (Station),
     CONSTRAINT sensorFK
         FOREIGN KEY (sensor)
         REFERENCES sensor (sensor)
  );
```

# Review

THE UNIVERSITY OF SYDNEY

# Reprise Participation Marking

**Requirements**

- Submit code at end of each week
- **Jupyter Notebooks:**
  - The various exercises have placeholder cells marked as TODO:

```
# TODO: replace the content of this cell
raise NotImplementedError
```

  - The content of these cells needs to be replaced with your own solution => basis for participation marking

**Output**

- Code/spreadsheets from exercises

**Marking**

- 10% of overall mark
- each week's participation assessed as: all done, partially done, no participation

***Submit your Jupyter notebooks in Canvas by next Friday***

# Tips and Tricks

- Real data is 'dirty' – **data cleaning** and transformation essential
- Database systems are great for *shared*, *persistent* storage of *structured* data, and also for consistent updating ('life' data)
- But some caveats:
  - Schema-first
  - Relational model quite restrictive (1NF, no lists, collections etc)
  - Not too intuitive; 1:1 mapping from spreadsheets doomed to fail
  - Type-mismatches between programming languages and SQL
  - Needs to be installed and maintained
    (though much better nowadays for SQLite and PostgreSQL)
- What's the benefit?
  - Sharing! Large data sets! Querying will give us some leverage too.

# Next Time

# Next Lecture : 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 23

**Exercises**

- [TODO]

**TODO in W5**

- Finish Grok Python modules
- Finish Grok SQL modules
- project data

# Many Good Python Resources

- Hard to make recommendations given different backgrounds
- Look online, there are many free resources and example code
- A few lists:
  - https://www.fullstackpython.com/best-python-resources.html
  - https://www.quora.com/Learning-Python/How-should-I-start-learning-Python-1

# Project Stage 1

# Project Stage 1: Explore, Clean, Pitch

**Objective**

Explore a data set and define a research question based on research/business requirement.

**Activities**

- Choose a data set
- Explore and summarise data set
- Clean and prepare data
- Define problem

**Output**

- **2-page report** summarising data, problem and explorative analysis
  - how did you acquire the data?
  - which tools did you use to clean and explore the data set?
  - with title page & references: max 4p

**Marking**

- 13% of overall mark

# SUGGESTED Timeline for Project Stage 1

- W1: Identify possible data sets
- W2: Identify and Explore possible data sets
- W3: Select project data set, define problem, complete exploration
- ***W4: Draft summary (problem & exploratory analysis)***
- W5: Clean and prepare data
- W6: Descriptive Stats, justification of suitability