Python with Database

Library: pandas, sqlite, pyodbc, csv

import pandas as pd

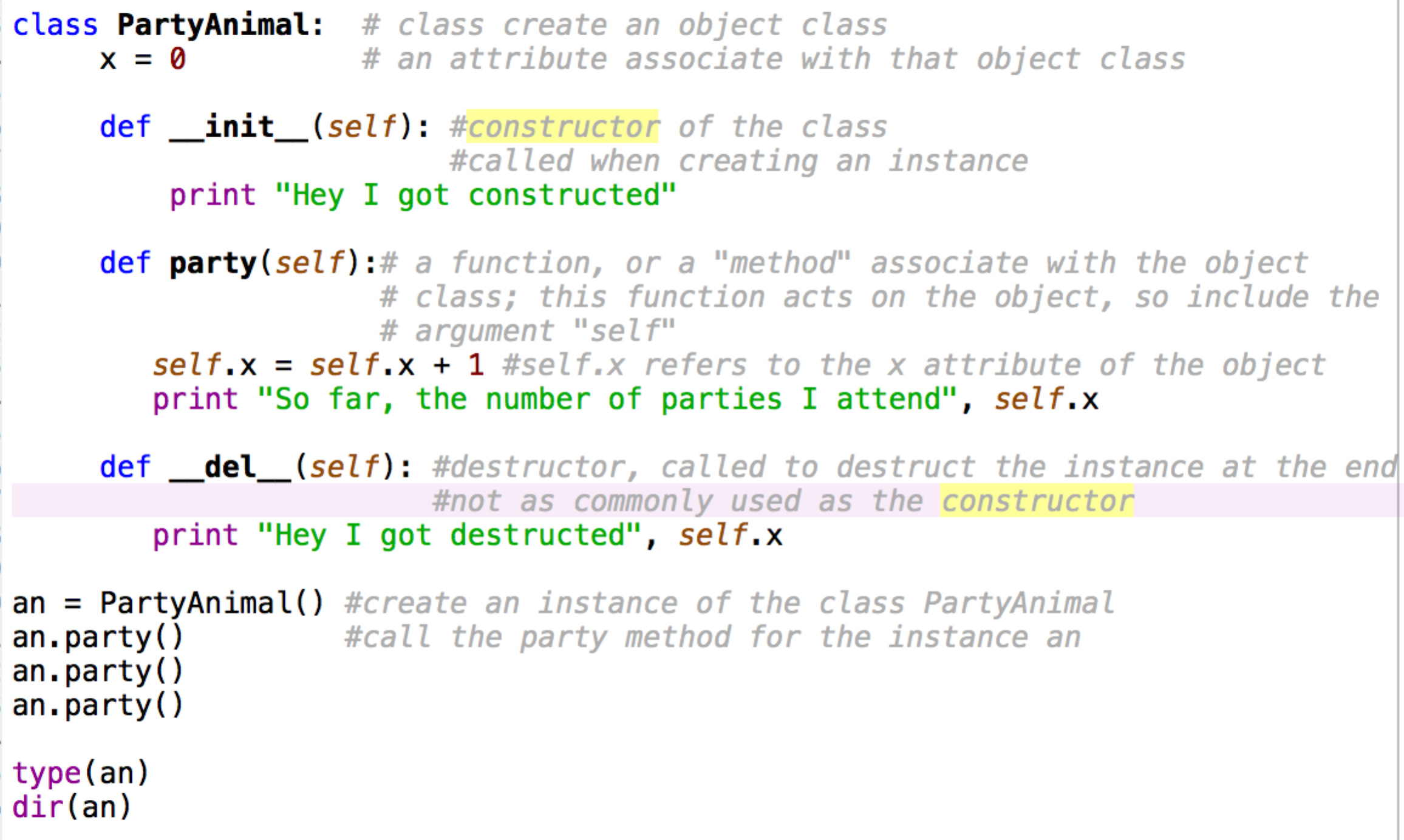
import sqlite3

import pyodbc

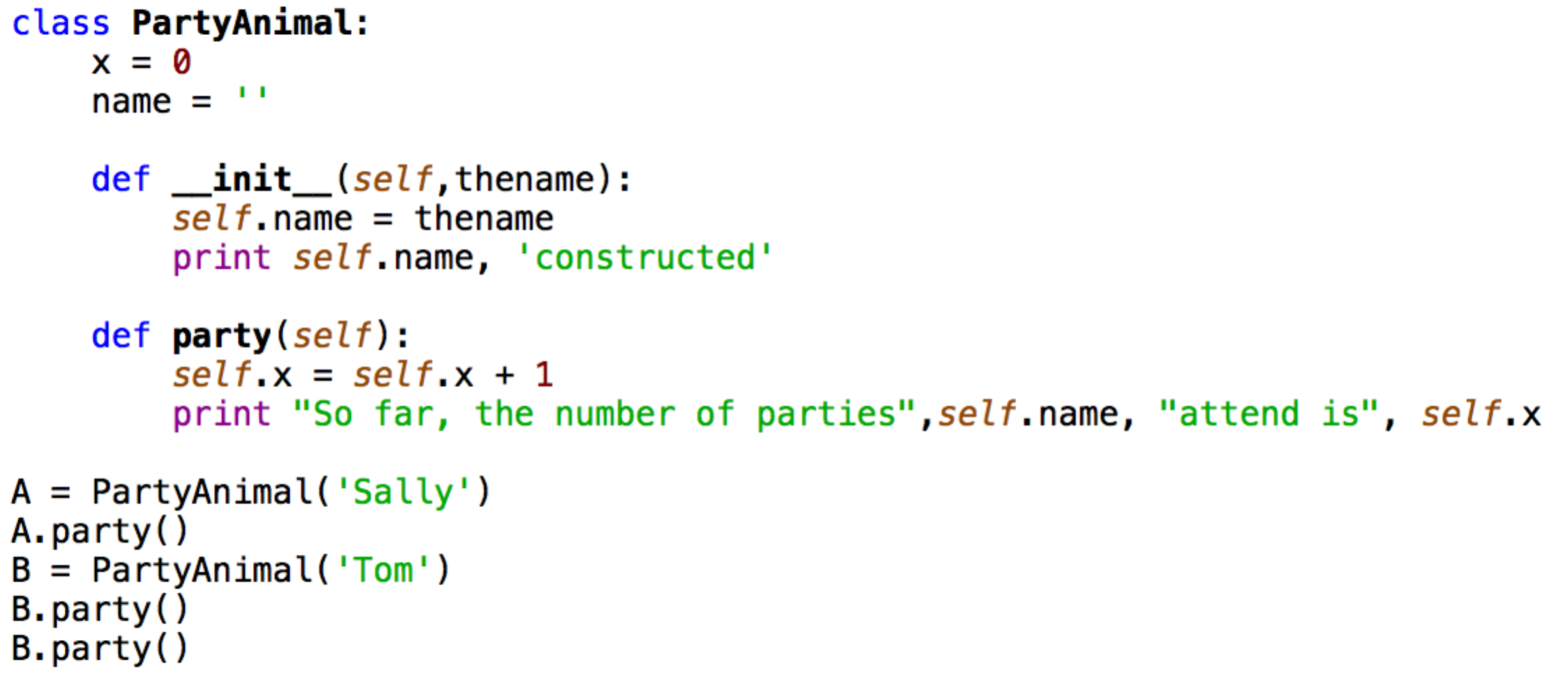
import csv

1. Object Oriented Python: OO programming

class, instance, method, field/attribute



Adding attributes when constructing an instance:



Inheritance: parent class – child class

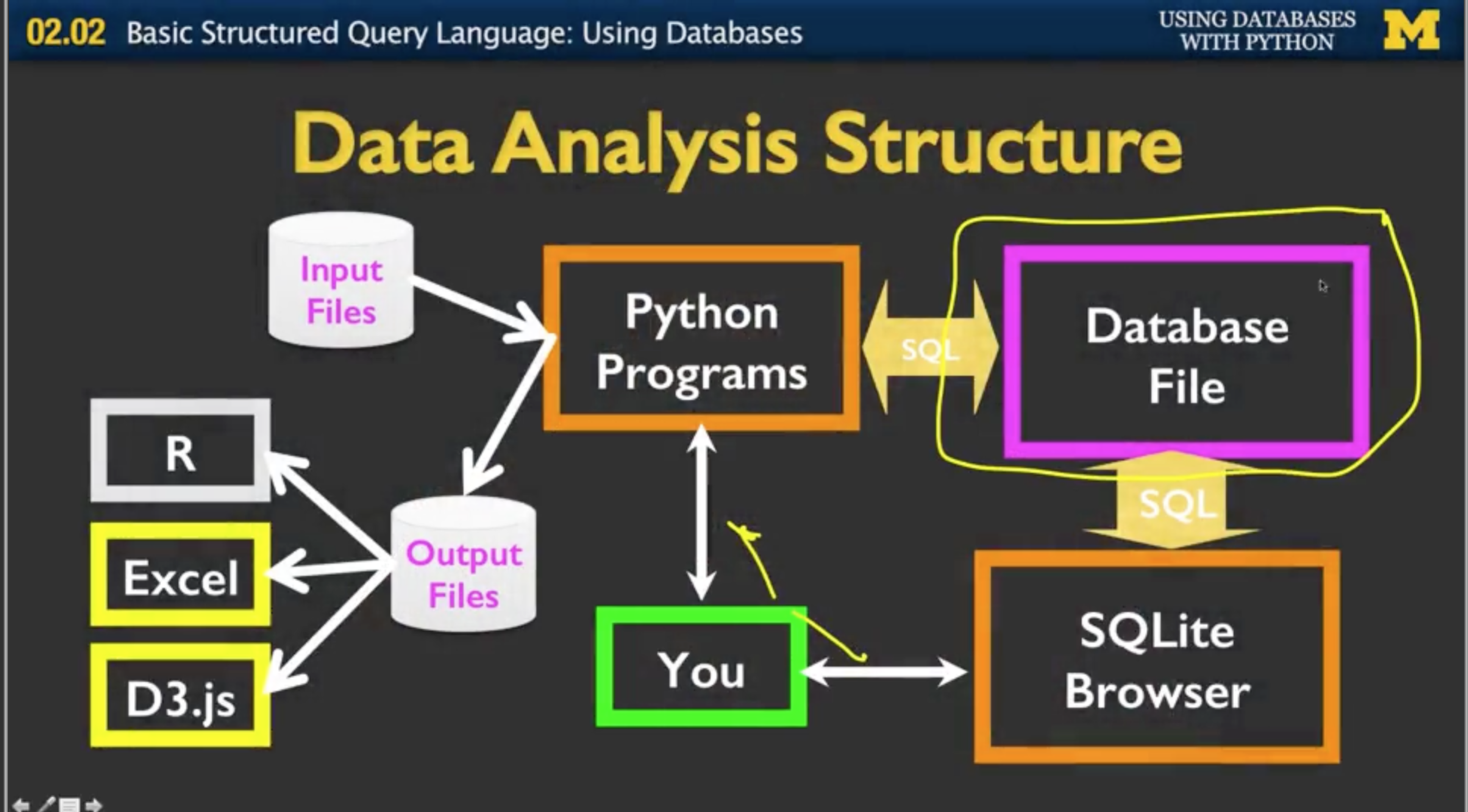


1. Basic SQL: single-table

Database ⊇ Relation (table) ⊇ Tuple (row) & Attribute (column)

APP (using Python, Java, etc.) <----> API (interface, using SQL) <----> database

In large project structure, there’s role of application developer that writes program that modify the application software, and uses SQL to get data from database, but cannot change the data in the database directly (the top-left role in the figure); and the role of database administrator (DBA), that talks to database directly using SQL through database systems (the lower-right role in the figure).



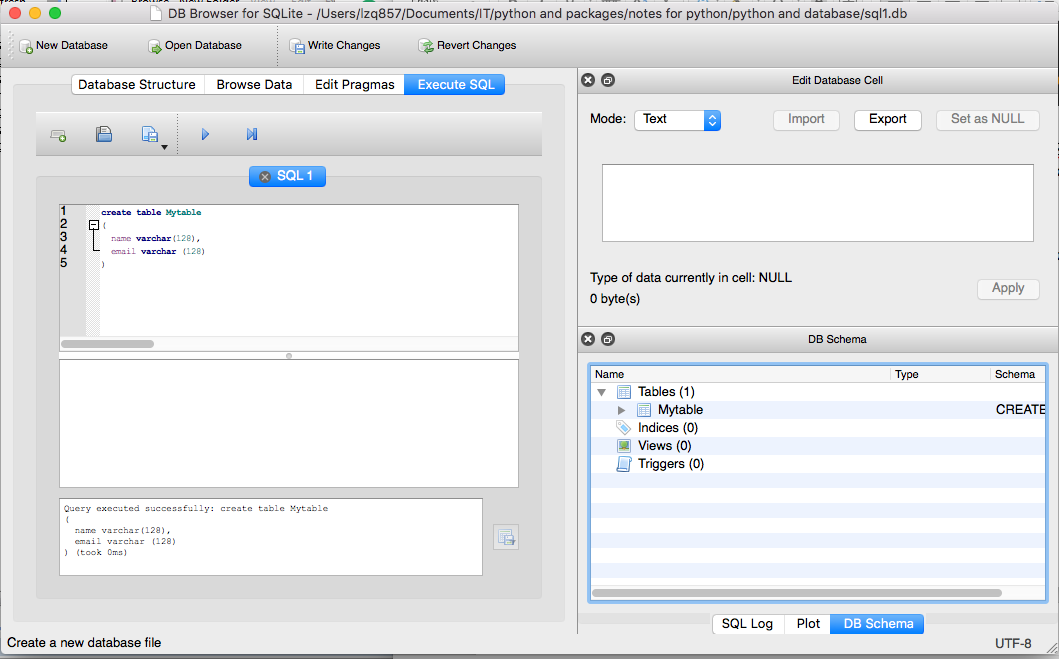
Database system talk to the database: e.g. Oracle, Mysql, SqlServer, SQLite browser by Firefox - SQLite manager, or SQLite browser download from <http://sqlitebrowser.org/>

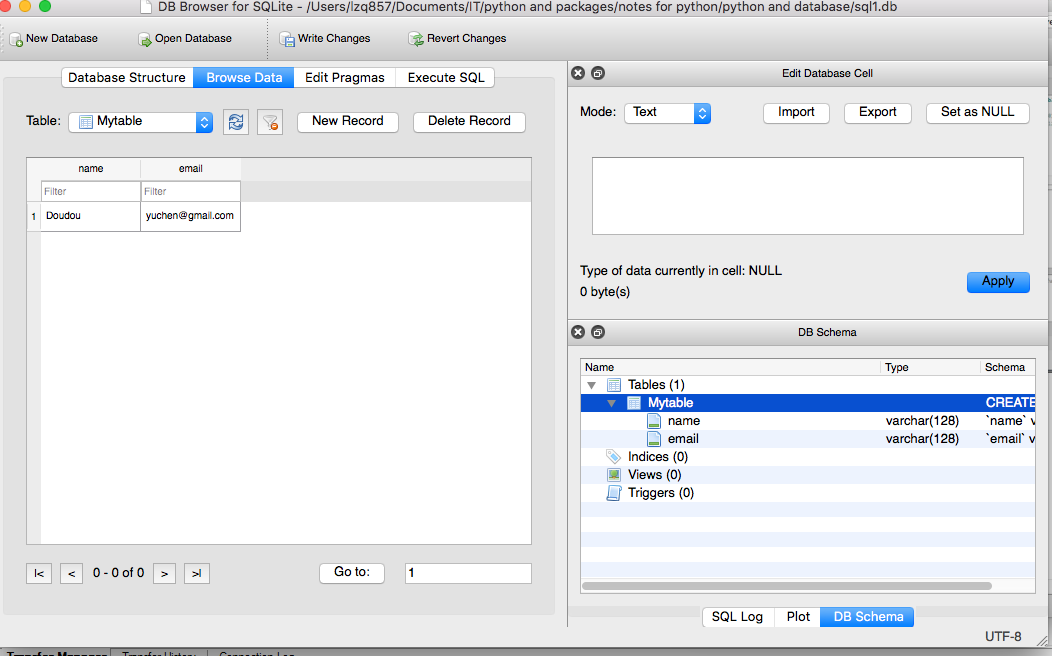
The created (new) database is of extension .db

Create new database or open existing database;

Run SQL in Execute SQL tab: enter query and run (shortcut F5 or ctrl+enter)

Check current content in Browse Data tab; new row can be input like in Excel by hitting “New Record” and input in the cells, or can be deleted (the corresponding queries can be seen by clicking the “SQL Log” tab)





Basic SQL operations (CRUD: create, read, update, delete)

* + Create: create an empty table in the database with various columns

CREATE TABLE table\_name (column1\_name column1\_type, column2\_name column2\_type);

/\* column names and types are the schema of the table

e.g.

CREATE TABLE Users (

name VARCHER(128),

email VARCHER(128)

)

/\* VARCHER refers to variable length character, 128 is the maximum character length; other types include INTEGER

When constructing a table, choose one column to be the primary key

Just in case table already exist, drop table before create, e.g.

DROP TABLE IF EXISTS Table1;

CREATE TABLE Table1(…)

Or:

CREATE TABLE if not exists Table1(…)

A table could also be created by the GUI using Edit->Create Table->input name and add fields…-> check ‘PK’ for primary key, check ‘AI’ for automatically increment (this will automatically be filled each time a new row is created), ‘U’ is for unique

* + Insert:

INSERT INTO table\_name (column1, column2) VALUES (val1, val2);

# This inserts a new row into the table, with these values in each column

e.g.

INSERT INTO Users (name, email) VALUES (‘doudou’, ‘doudou@umd.edu’)

* + Delete

DELETE FROM table\_name WHERE column1 = val1

/\* This deletes an existing row from the table, whose column has specified value

e.g.

DELETE FROM Mytable WHERE name = 'huairen'

* + Update: change the information of some particular cells

UPDATE table\_name SET column1 = val1 WHERE column2 = val2

e.g.

UPDATE Users SET name = ‘Charles’ WHERE email = ‘csev@umich.edu’

/\* This changes the name cell into Charles whose email is as specified

/\* Without WHERE, all rows will be updated

Combine insert and update:

INSERT OR REPLACE INTO Users (name, email) VALUES (‘doudou’, ‘doudou@umd.edu’)

/\* This insert into Users if doudou does not exist, or update it if exists \*/

* + Select: return a list of columns
* SELECT \* FROM table\_name WHERE column1 = val1

SELECT column\_names FROM table\_name WHERE column1 = val1

/\* ‘\*’ means all columns; without WHERE+condition, just select all rows

/\* To select some particular columns, use the columns’ names in place of \*

e.g.

SELECT \* FROM Users WHERE email = ‘doudou@gmail.com’

SELECT name, age FROM Users WHERE email = ‘doudou@gmail.edu’

* SELECT \* FROM table\_name ORDER BY column1 DESC/ASC

/\* Select the rows with descending/ascending order in column1

e.g.

SELECT \* FROM Users ORDER BY name DESC

* In general

SELECT <columns> FROM <table> WHERE <condition> ORDER BY <column> DESC/ASC LIMIT 20 /\* LIMIT 20 displays the first 20 rows

Python SQL package: pyodbc and Giraffez (to talk to Teradata), pyscopg (to talk to Redshift), sqlite, and pandas

* Connect to the database (check python cmd for set-up)

import sqlite3

conn = sqlite3.connect('databasename.sqlite')

*#For pyodbc <---> Teradata, use*

conn = pyodbc.connect('DRIVER={Teradata};DBCNAME=oneview;UID=lzq857;PWD=Smile123;', autocommit=True)

*#No hardcode EID/PASSWD connection:*

import pyodbc

import getpass

user = getpass.getuser()

pwd = getpass.getpass()

conn = pyodbc.connect('DRIVER={Teradata};DBCNAME=oneview;UID=' +user + ';PWD=' + pwd+';',autocommit=True)

conn.commit() *#commit changes to the original database*

conn.close() *#close the connection*

* Connect to Redshift via pyscopg

*#Connecting to the Redshift instance*

import psycopg2

connection\_string = "dbname='Redshift\_DB' port='5439' user=LZQ857' password='Baodou11' host=' pbcdwp.cloud.capitalone.com' sslmode='require'"; *#the host could always change*

conn = psycopg2.connect(connection\_string);

*#Alternatively, no hardcode userID/password connection using the predefined .tdlogon file:*

import psycopg2

import getpass

import sys

from os.path import expanduser

user = getpass.getuser()

home = expanduser("~")

f = open(home + '/.tdlogon')

*#alternatively, use f = open(‘/prod/user/home/’ + user + '/.tdlogon' on statgw)*

logline = f.readline()

usr = logline.split(',')[0].split('/')[1].strip()

pwd = logline.split(',')[1].strip()

conn = psycopg2.connect(dbname='db', host='pbcdwp.cloud.capitalone.com', port=5439, user=usr, password=pwd)

*#Importing data from Redshift tables into a pandas dataframe*

import pandas as pd

dataDF = pd.read\_sql("select top 10 \* from tablename", conn)

* Cursor class

cursor = conn.cursor() *#cursor represents the database to be executed*

cursor.commit() *#commit changes to the table, say in Teradata*

cursor.close() *#close cursor when everything is done*

* Execute SQL statement to cursor

cursor.execute(''' SQL statement ''')

or

cursor.executescript(''' SQL statements ''')

*#the executescript() method is able to execute many statements in a row, separated by ;*

e.g.

cursor.execute(''' DROP TABLE IF EXISTS table\_name''')

*#execute this to delete the table in the database prior to create*

Other way:

sqlCreate = ''' SQL create statement'''

sqlSelect = ''' SQL select statement '''

cursor.execute(sqlCreate), cursor.execute(sqlSelect)

In general, the SQL query may need value from python variables, as

cursor.execute(''' SQL statement ''', python\_variable)

e.g.

cursor.execute(''' INSERT INTO Counts (org, count) VALUES ( ?, 1 ) ''', ( org, ))

*#The statement will take the value from python variable “org” and insert its value to the place marked by “?” in the query*

* Show the result of a query on the cursor table

cursor.fetchone()

*#return the next row of the resulting cursor table after the previous query, return None if no more rows are available*

e.g.

cursor.execute(''' CREATE TABLE temp as (select … from …) ''')

row = cursor.fetchone()

while row is not None:

print(row)

row = cursor.fetchone()

*#will print every row in the table created*

cursor.fetchall()

*#All rows remaining in the result set from the query*

cursor.fetchmany(size = 10)

*#The first ten rows of the result set*

Note: cursor.fetch only displays the resulting set as a result of acting the previous SQL query on the cursor table, while cursor.commit preserves the changes to the table in the database.

* cursor.description

#Return a list of tuple, [(column\_name, type, …), …]

* pd.read\_sql(''' SQL query ''', conn)

df = pd.read\_sql(''' SQL query''', conn)

*#Read SQL query and database into a dataframe*

e.g.

df1 = pd.read\_sql(''' select \* from ud206.lzq857\_table\_name ''', conn)

df1.to\_csv('path/table\_name.csv')

*#Save df1 as the csv file*

* Output the cursor as a .csv file: csv\_writer

import csv

import time

timestr = time.strftime("%Y%m%d-%H%M%S")

out\_file = timestr + 'onem\_cca\_core\_vw.csv'

with open(out\_file, 'w') as fname:

csv\_writer = csv.writer(fname, quoting=csv.QUOTE\_MINIMAL)

*#create csv\_writer, output the result to out\_file*

csv\_writer.writerow([i[0] for i in cursor.description])

csv\_writer.writerows(cursor)

* Pandas can read directly from excel, e.g.

import pandas as pd

df\_table1 = pd.read\_excel('table1.xlsx')

df\_table2 = pd.read\_excel('table2.xlsx')

df\_sum = pd.merge(df\_table1, df\_table2, on='ID', how='left')

df\_sum.to\_excel('joined\_table.xlsx')

* Use csv package to read/write to .csv file, e.g.

import csv

with open(‘some.csv’, ‘wb’) as f\_csv: # ‘rb’ to read, b means binary data

list\_csv = csv.DictReader(f\_csv) # read the csv file as a list of dictionary

reader = csv.reader(f\_csv, delimiter = ‘|’) # sometime try delimiter = ' '

writer = csv.writer(f\_csv)

for data in list1:

writer.writerow(data)

* Push back to database

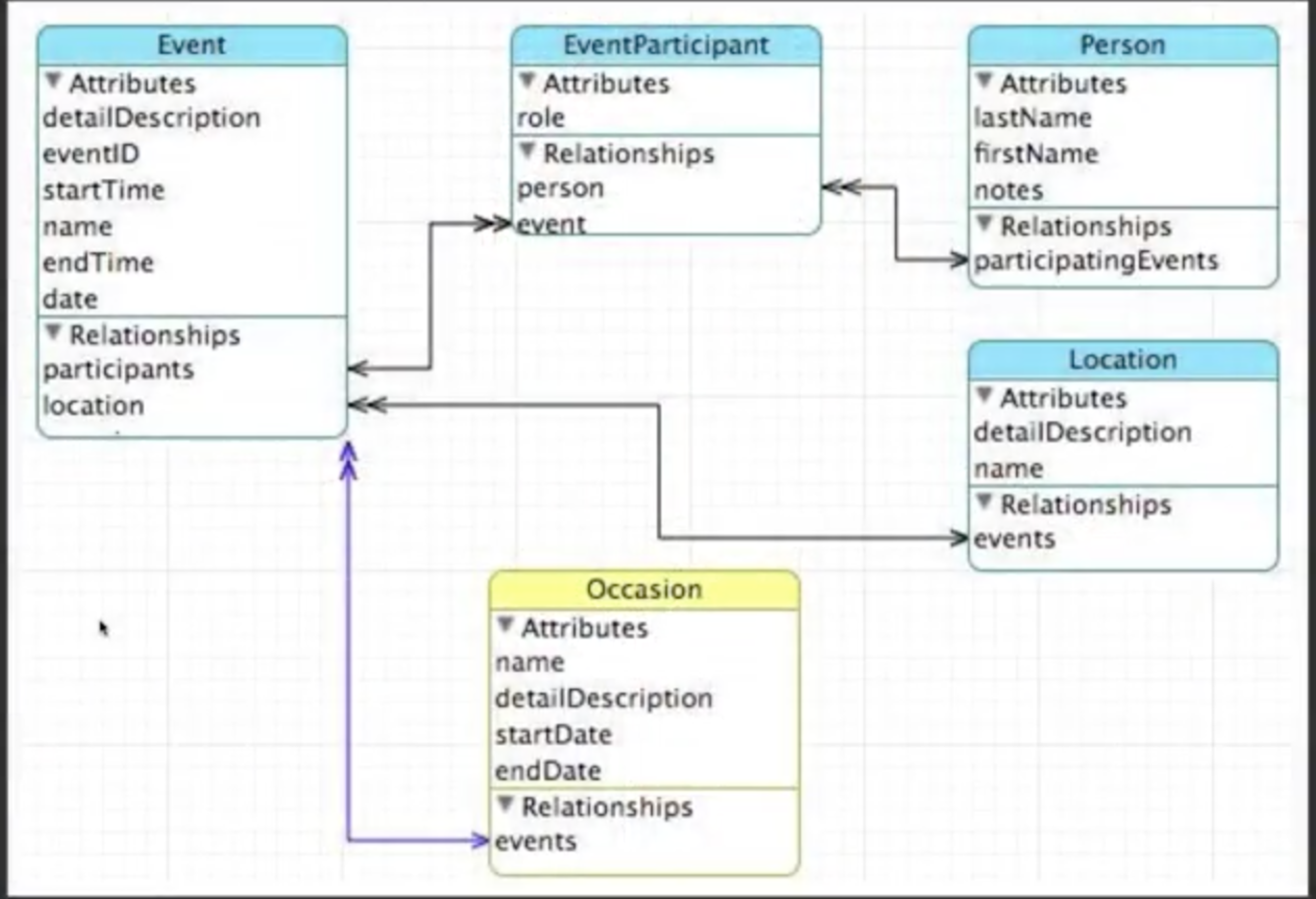
df1.to\_sql("table\_name", conn, in\_exists="replace")

1. Data model design and relational SQL: multi-table relational SQL

Within a data model there are various tables each with multiple tables, and there are some columns specifically to connect one table to another.

1st task before build a data model: draw a logical picture to capture multiple tables—the primary column, the attributes columns for each table, and the connections between multiple tables; then map the logical picture to the physical picture of databases.

Principle: do not have replicate string in a table – use multiple tables and connect them.



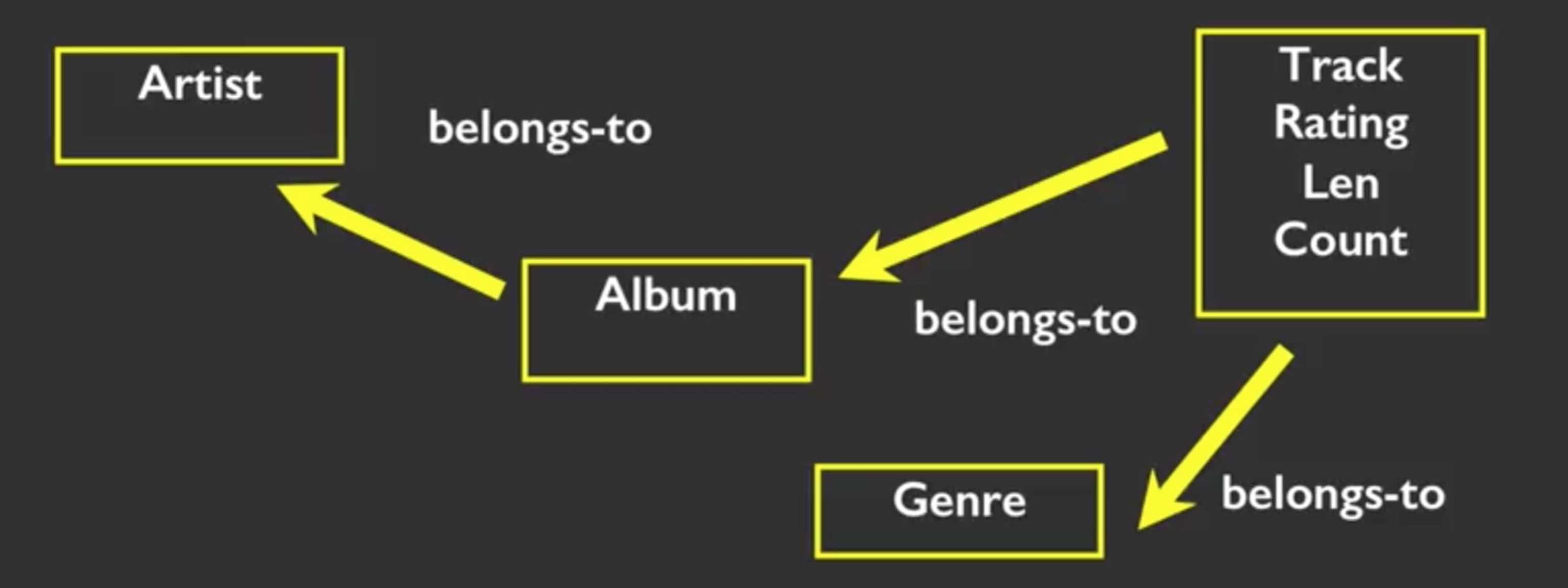
e.g. build an application for music track managing.

* Start with an Excel for all the data:



* 1st: draw the logical picture.

The first table to draw in the picture is the one that contains the focused information of the project: the track table, containing track names, length column, and rating column. Then build an album table and relate that to the track table – as track belongs to an album. Then an artist table and relate that to the album table. Then a genre table and relate it to the track as well.

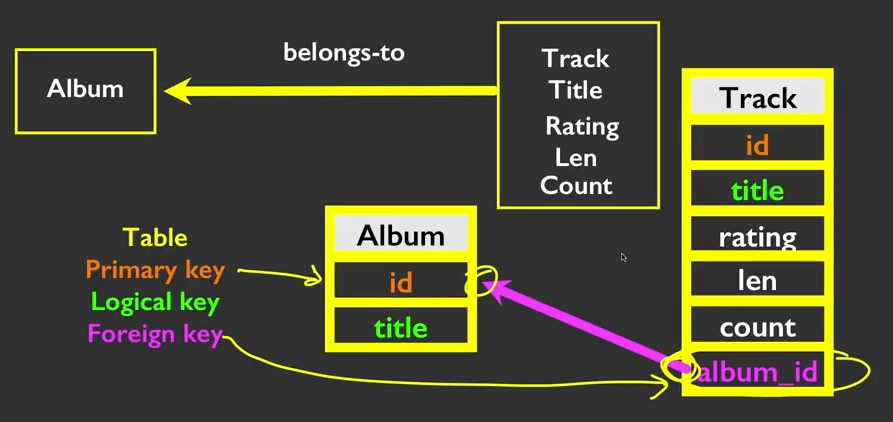


* Next we map the logical picture to databases: take each of the tables in the logical picture and the relations between the tables, and add some information.

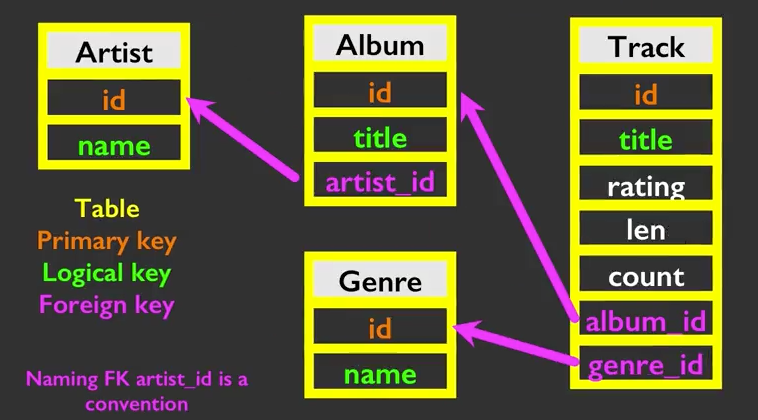
In the track table we add an ID column as the primary column (primary key is a unique number for each row, so that a particular row could be referred), and we have columns like track name, length, rating, and we add an album\_id column so we can relate the track table to the album table.

In album table we have id — primary column, and title. The album id is related to the album\_id column in the track table. The column title is sometimes referred as the logical key used to look up the row from outside.

Here, the album\_id column in the track table is a foreign key (the starter of an arrow of relation)



Keep doing this for the other tables we end up with



Now creating the physical tables; don’t forget to specify the primary key when creating the table.

e.g.

CREATE TABLE Artist (

`id` INTEGER NOTE NULL PRIMARY KEY AUTOINCREMENT UNIQUE,

`name` TEXT

)

After creating the physical tables, just insert the relational data into the table.

When inserting data for a foreign key, say, when inputting data in the artist\_id column for the Album table, don’t directly putting the artists’ name – instead, put the id associated with an artist’s name in the Artist table – that’s the spirit of this multi-table structure.

Next, perform join on tables that are connected by an arrow.

* + Join: link two tables based on their common

SELECT <table.columns> FROM <left\_table> JOIN <right\_table> ON <predicate>

/\* left\_table refer to the head of arrow, and right\_table the tail of arrow

e.g.

SELECT Album.title, Artist.name FROM Album JOIN Artist ON Album.artist\_id = Aritst.id

e.g. XML to Database

In itunes, go files -> library -> export library, one can export the itune library as a XML format file.

XML has a “dictionary over dictionary over dictionary” structure. Use package xml.etree.ElementTree to get the entry from the XML file.

import xml.etree.ElementTree as ET

import sqlite3

fname = '~\python\Library.xml'

stuff = ET.parse(fname)

all = stuff.findall('dict/dict/dict')

# this find all entries at the third level of dictionary, each entry like ‘<format>entry value</format>’

# e.g. <key>Artist</key>, <string>Queen</string>

# We can loop through the entries in all to read in data, e.g.

def lookup(entry, column):

found = False

for child\_entry in entry:

if found : return child\_entry.text

if child\_entry.tag == 'key' and child\_entry.text == column:

found = True

return None

Create the tables in the logic picture, include the foreign keys in the table (if there is):

cur.executescript('''

CREATE TABLE IF NOT EXISTS Artist(

id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,

name TEXT UNIQUE

);

CREATE TABLE Genre (

id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,

name TEXT UNIQUE

);

CREATE TABLE Album (

id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,

artist\_id INTEGER,

title TEXT UNIQUE

);

CREATE TABLE Track (

id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,

title TEXT UNIQUE,

album\_id INTEGER, genre\_id INTEGER,

len INTEGER, rating INTEGER, count INTEGER

);

''')

Loop through the XML file and insert into table the values:

for entry in all:

if ( lookup(entry, 'Track ID') is None ) : continue

name = lookup(entry, 'Name')

genre = lookup(entry, 'Genre')

artist = lookup(entry, 'Artist')

album = lookup(entry, 'Album')

count = lookup(entry, 'Play Count')

rating = lookup(entry, 'Rating')

length = lookup(entry, 'Total Time')

if name is None or artist is None or album is None :

continue

print name, artist, album, count, rating, length

cur.execute('''INSERT OR IGNORE INTO Artist (name)

VALUES ( ? )''', ( artist, ) )

cur.execute('SELECT id FROM Artist WHERE name = ? ', (artist, ))

artist\_id = cur.fetchone()[0]

cur.execute('''INSERT OR IGNORE INTO Album (title, artist\_id)

VALUES ( ?, ? )''', ( album, artist\_id ) )

cur.execute('SELECT id FROM Album WHERE title = ? ', (album, ))

album\_id = cur.fetchone()[0]

cur.execute('''INSERT OR IGNORE INTO Genre (name)

VALUES ( ? )''', ( genre, ) )

cur.execute('SELECT id FROM Genre WHERE name = ? ', (genre, ))

genre\_id = cur.fetchone()[0]

cur.execute('''INSERT OR REPLACE INTO Track

(title, album\_id, len, rating, count, genre\_id)

VALUES ( ?, ?, ?, ?, ?, ? )''',

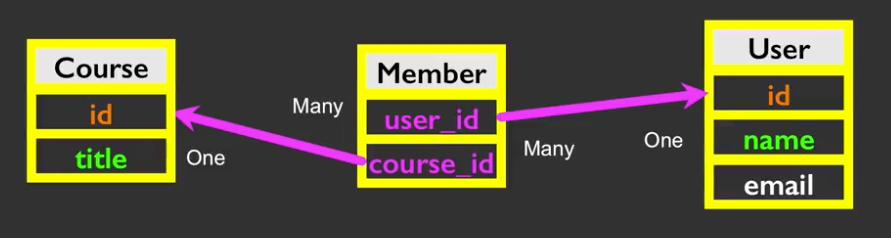
( name, album\_id, length, rating, count, genre\_id ) )

conn.commit()

1. Many-to-many relationships in SQL

We have been modeling many-to-one databases, such as the previous case – an Album points to many Tracks, but not otherwise. So we can put a foreign key of Album\_id in the Tracks table that points to the Album the track belongs. Sometimes we need to model many-to-many relationships, and this cannot be modeled with a single foreign key. An example for this is Students <-> Courses: we can have different students registered for one course, and we can have one students registered for many courses.

In this case, we put an intermediate table in between, that contains both foreign keys as primary key, and then we can model both side as a one-to-many relationship. See below.



CREATE TABLE IF NOT EXISTS Course(

id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,

title TEXT

);

CREATE TABLE IF NOT EXISTS User(

id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,

name TEXT,

email TEXT

);

CREATE TABLE IF NOT EXISTS Member(

user\_id INTEGER,

course\_id INTEGER,

PRIMARY KEY (user\_id, course\_id)

);

/\* Insert data, omitted… \*/

SELECT User.name, Member.role, Course.title

FROM User JOIN Member Join Course

ON Member.user\_id = User.id AND Member.course\_id = Course.id

ORDER BY Course.title, User.name

e.g. JSON to database

JSON is basically a list of lists/dicts. Use package json to read JSON by Python

import json

import sqlite3

try:

str\_data = open('data.json').read() # read the whole .json file into memory

except:

raise Exception('Error in read in the file')

json\_data = json.loads(str\_data)

# now we can loop through json\_data for each entry

Insert into the created table the data from the .json file:

for entry in json\_data:

name = entry[0]

title = entry [1]

cur.execute(''' INSERT OR IGNORE INTO User (name)

VALUES (?) ''', (name, ))

cur.execute(''' SELECT id FROM User WHERE name = ? ''', (name, ))

user\_id = cur.fetchone()[0]

cur.execute(''' INSERT OR IGNORE INTO Course (title)

VALUES (?) ''', (title, ))

cur.execute(''' SELECT id FROM Course WHERE title = ? ''', (title, ))

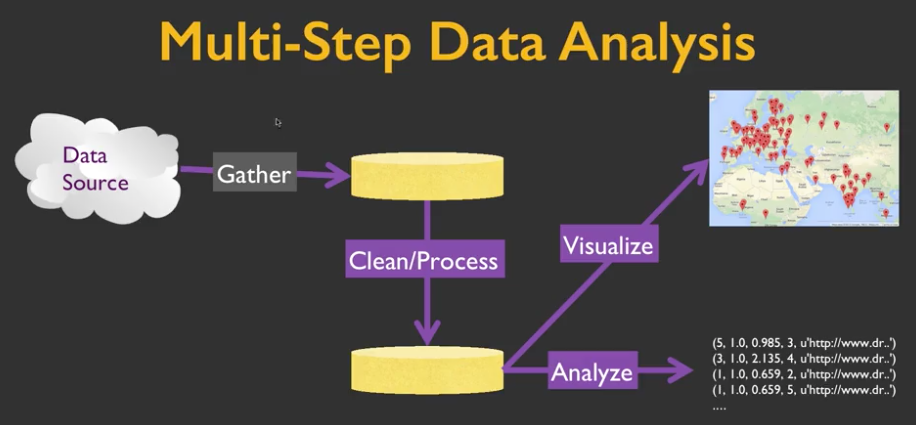
course\_id = cur.fetchone()[0]

cur.execute(''' INSERT OR REPLACE INTO Member (user\_id, course\_id)

VALUES (?, ?) ''', (user\_id, course\_id))

conn.commit()

1. Databases and Visualization: geocoding



Here in each step we write a separate python code, instead of putting everything together.