Master of Science in
Genomics and Bioinformatics

Python and Databases

GNBF5010 Lecture 11

2021R1

Overview

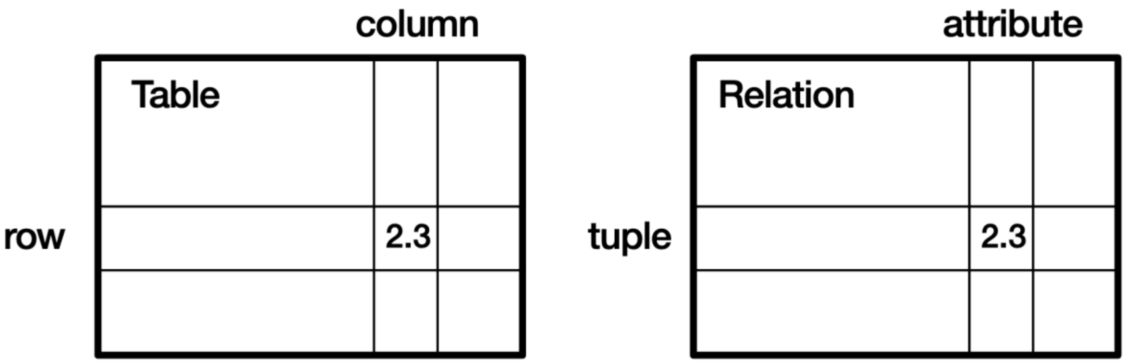
- Introduction to Databases and SQL

- Interacting with SQLite from Python

Introduction to Databases

What is a database

- An organized collection of structured information, or data, typically stored electronically in a computer system.
- Usually controlled by a database management system (DBMS)
  - e.g. MySQL, Microsoft SQL Server, Oracle Database, and **SQLite**.



- A relational database looks like a spreadsheet with multiple sheets (tables, or relations)

https://www.oracle.com/hk/database/what-is-database/
https://www.py4e.com/html3/15-database

Database vs Dictionary

- Both map from *keys* to *values.*
- Both are designed to keep the inserting and accessing of data very fast, even for large amounts of data.

- But a database is on a permanent storage, and thus persists after program ends.
- A database can store far more data than a dictionary.
  - A python dictionary is limited to the size of the computer memory.

https://www.py4e.com/html3/15-database

Database vs Spreadsheet

- Spreadsheets were originally designed for one user (or a few) who don't need to do a lot of complicated data manipulation.

- Databases are designed to hold much larger collections of organized information - massive amounts, sometimes.
- Databases allow multiple users at the same time to quickly and securely access and query the data using highly complex logic and language.

https://www.oracle.com/hk/database/what-is-database/

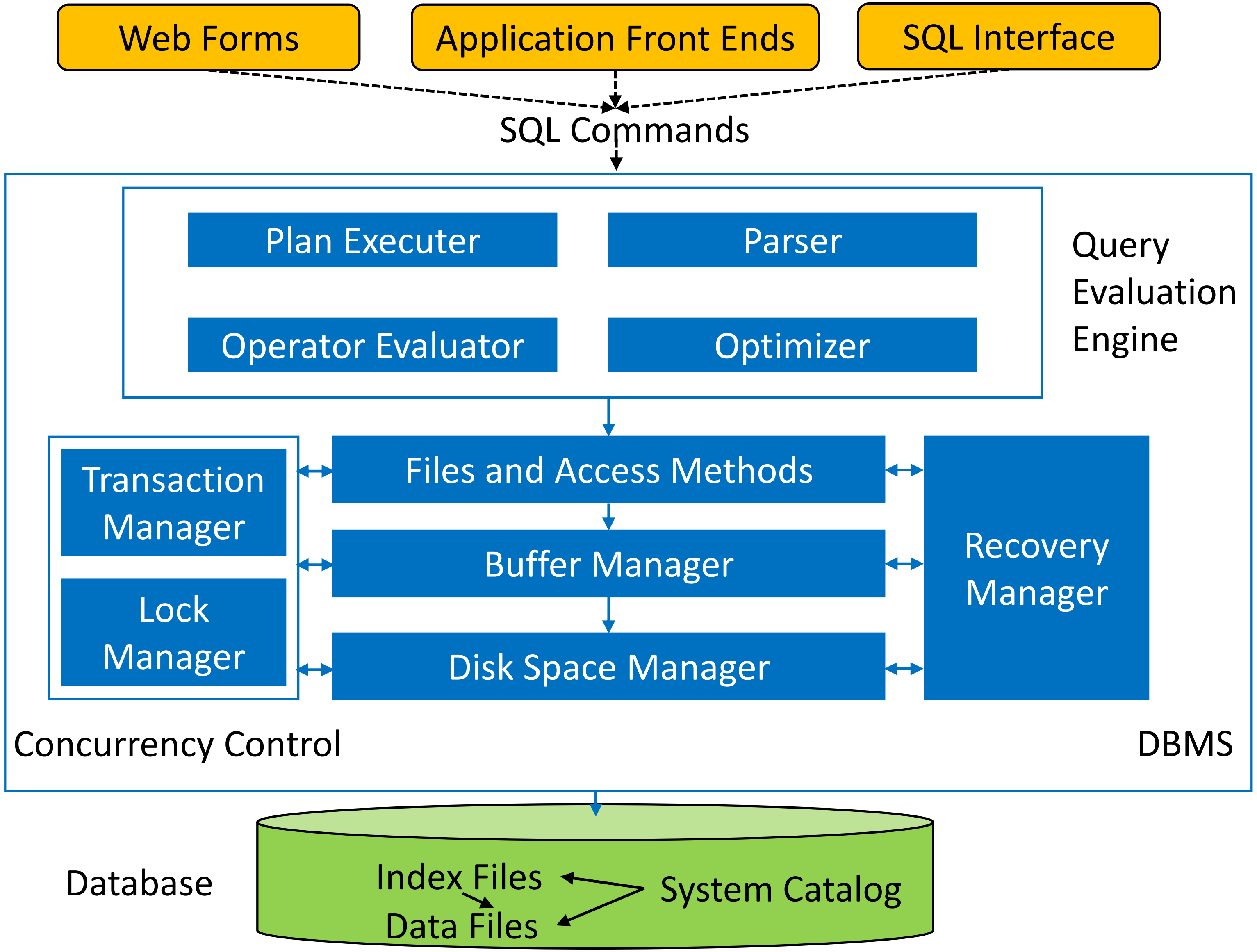
What is a database system

- A database system (or shortened to database)
  = **Data + DBMS + Applications**

- The data can then be easily accessed, managed, modified, updated, controlled, and organized.

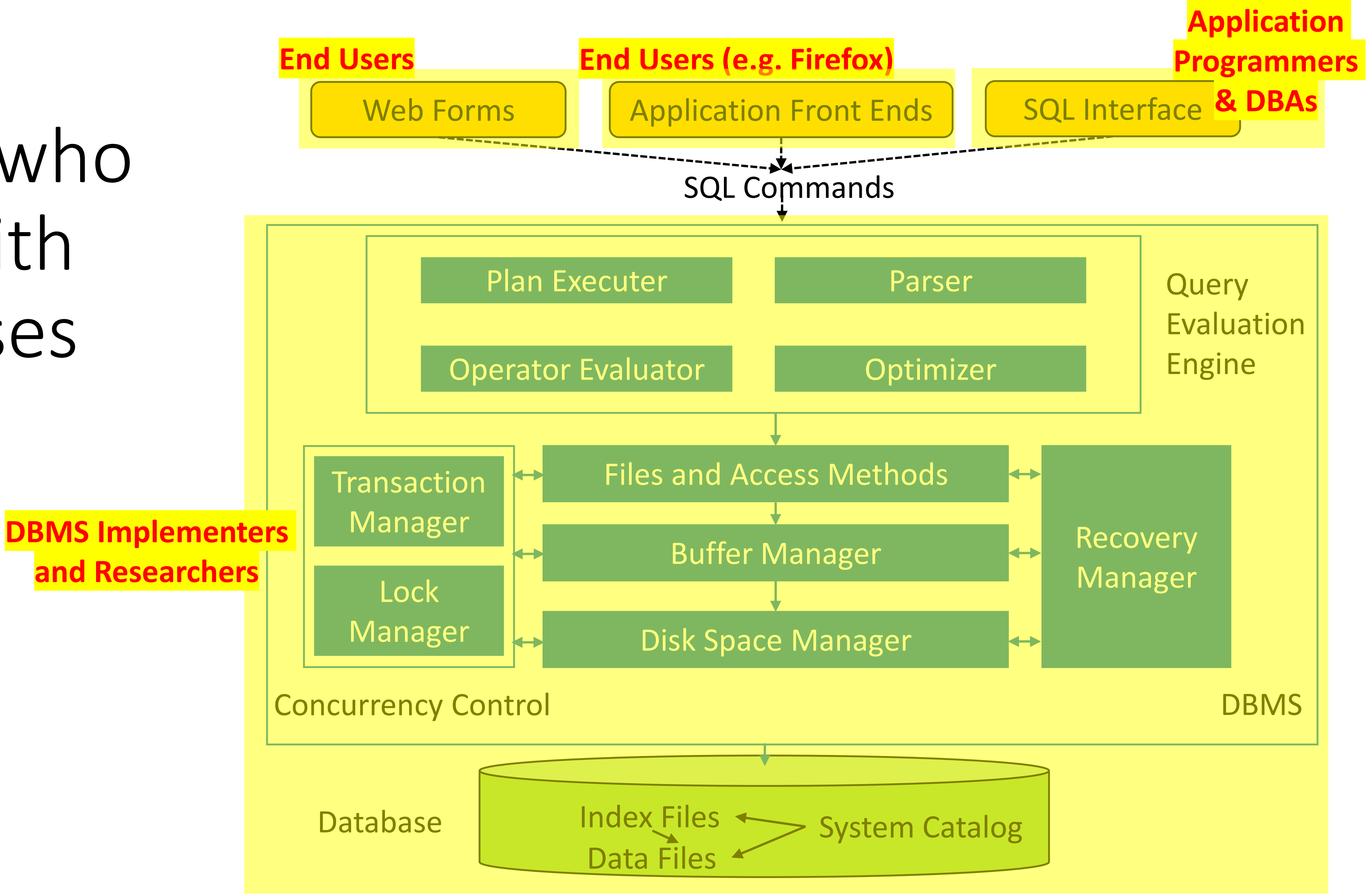
- Most databases use structured query language (SQL) for writing and querying data.

Architecture of a database system



https://web2.qatar.cmu.edu/~mhhammou/15415-s20/index.html

People who work with databases



https://web2.qatar.cmu.edu/~mhhammou/15415-s20/index.html

Common Database Systems

- Three major Database Management Systems (DBMS) in wide use
  - Oracle - Large, commercial, enterprise-scale, very very tweakable
  - MySql - Simpler but very fast and scalable, commercially open source
  - SqlServer - Very nice, from Microsoft

- Many other smaller projects, free and open source
  - HSQL, SQLite, Postgres, ...

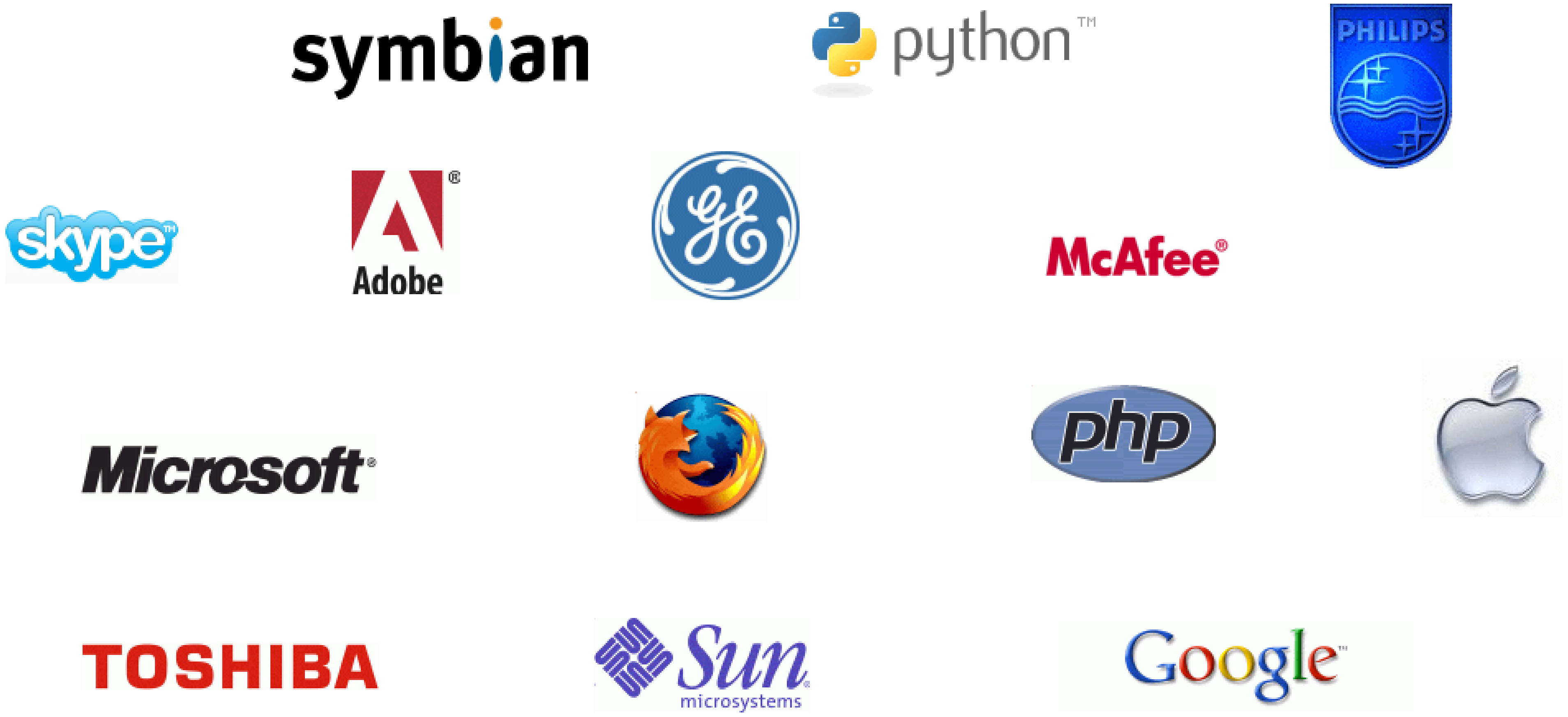
https://www.py4e.com/html3/15-database

The SQLite DBMS www.sqlite.org

- **SQLite** is widely-used and is favourite among the developers for many reasons:
- Extremely light-weighted (not more than 500 KBs)
- Serverless – don't need any separate server for availing its services
- No complex setup
- The data can be saved in a single file – easy to transfer

https://www.datacamp.com/community/tutorials/sqlite-in-python

SQLite is used by lots of software



http://www.sqlite.org/famous.html

DB Browser for SQLite

www.sqlitebrowser.org

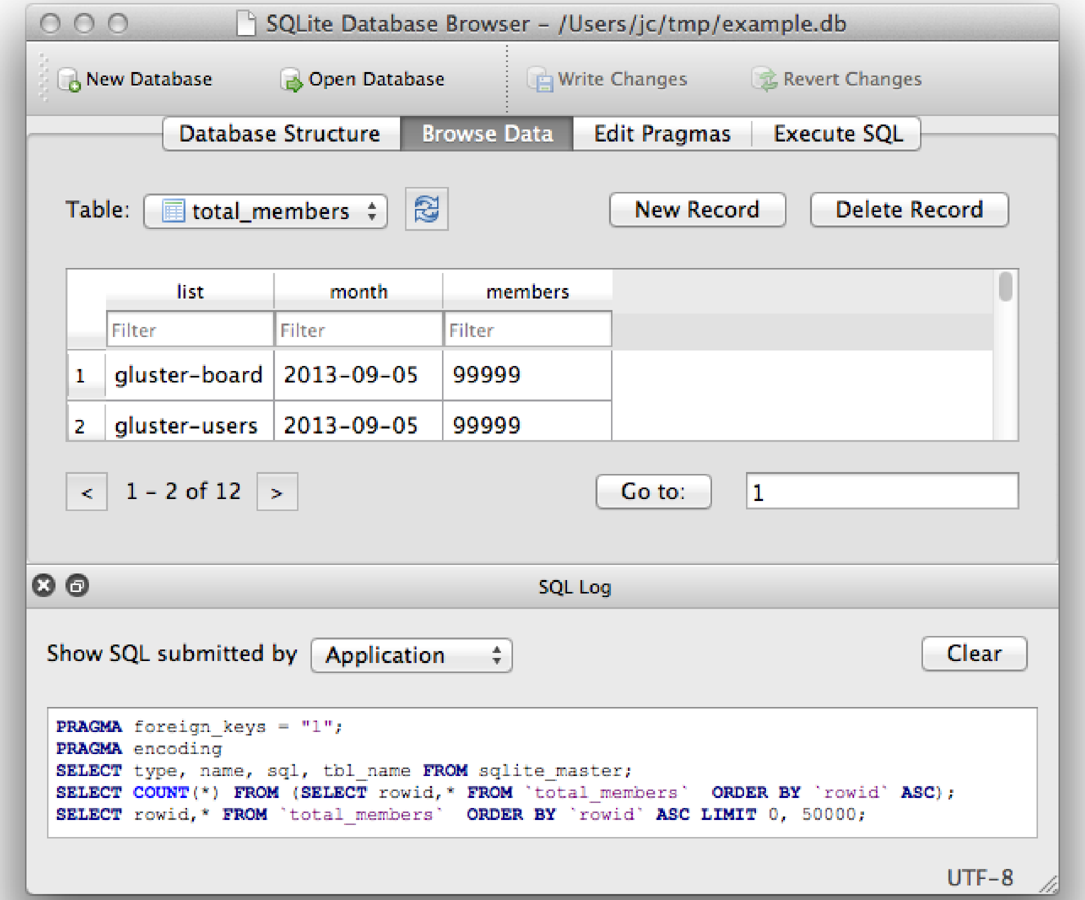


Table: total_members

	list	month	members
1	gluster-board	2013-09-05	99999
2	gluster-users	2013-09-05	99999

1 – 2 of 12

Go to: 1

Show SQL submitted by: Application

```
PRAGMA foreign_keys = "1";
PRAGMA encoding
SELECT type, name, sql, tbl_name FROM sqlite_master;
SELECT COUNT(*) FROM (SELECT rowid,* FROM `total_members`  ORDER BY `rowid` ASC);
SELECT rowid,* FROM `total_members`  ORDER BY `rowid` ASC LIMIT 0, 50000;
```

How to make a database

Structured Query Language (SQL)

The language we use to issue commands to the database

- Create data (a.k.a Insert)
- Retrieve data
- Update data
- Delete data

http://en.wikipedia.org/wiki/SQL

Create a database using SQLite

- In SQLite Browser, click "New Database" button.

- Alternatively, in command line, run

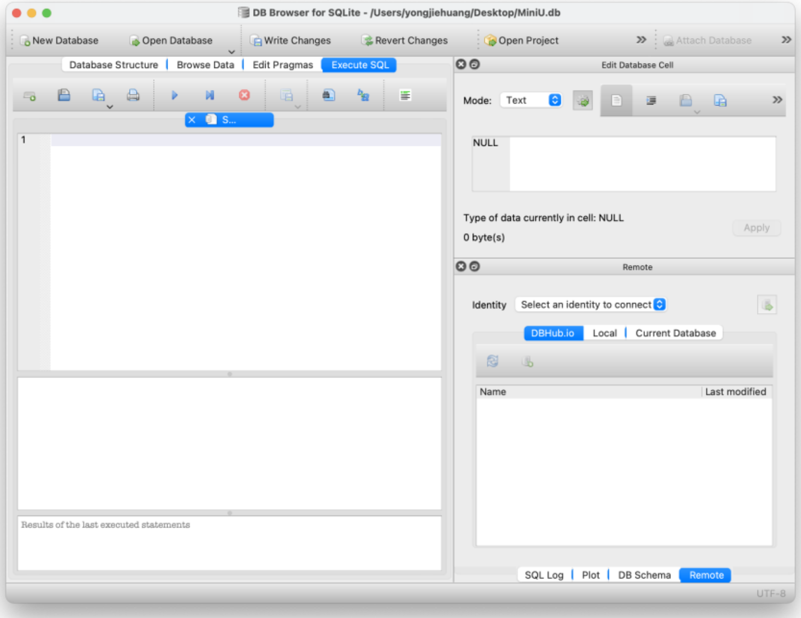
  ```
  $ sqlite3 <db_file>
  ```

  which will enter SQLite command line. To exit, type `.exit`
  See more at https://tool.oschina.net/uploads/apidocs/sqlite/sqlite.html

Create tables

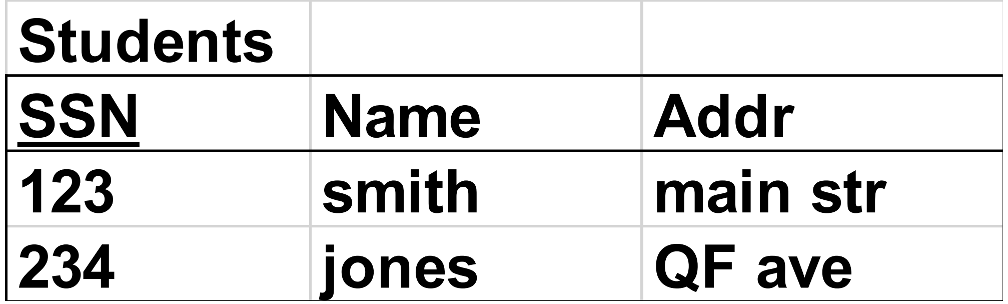
```
CREATE TABLE Students (
        SSN     INTEGER PRIMARY KEY,
        Name    TEXT,
        Addr    TEXT
);
INSERT INTO Students Values(123, "smith", "main str");
INSERT INTO Students Values(456, "jones", "QF ave");


CREATE TABLE Classes (
        CId     TEXT PRIMARY KEY,
        CName   TEXT,
        Units   INTEGER
);
INSERT INTO Classes Values("15-413", "s.e.", 2);
INSERT INTO Classes Values("15-412", "o.s.", 2);


CREATE TABLE "Takes" (
        "SSN"   INTEGER,
        "CId"   TEXT,
        "Grade" TEXT,
        FOREIGN KEY("CId") REFERENCES "Classes"("CId"),
        FOREIGN KEY("SSN") REFERENCES "Students"("SSN")
);
INSERT INTO Takes Values(123, "15-413", "A");
INSERT INTO Takes Values(456, "15-413", "B");
```

lec11-create-miniU.sql

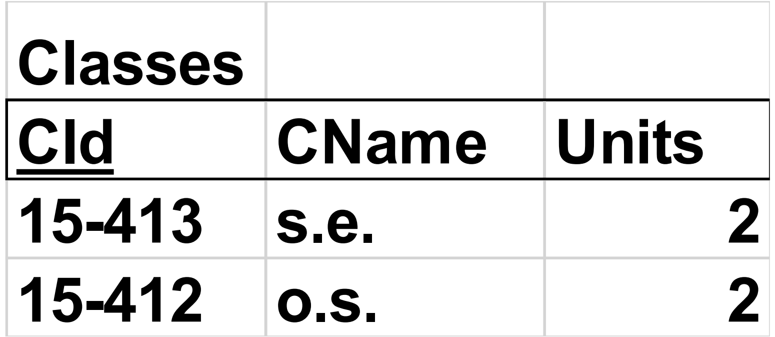


Execute commands in "Execute SQL" box in SQLite Browser; or run from command line (`$ sqlite3 MiniU.db`)

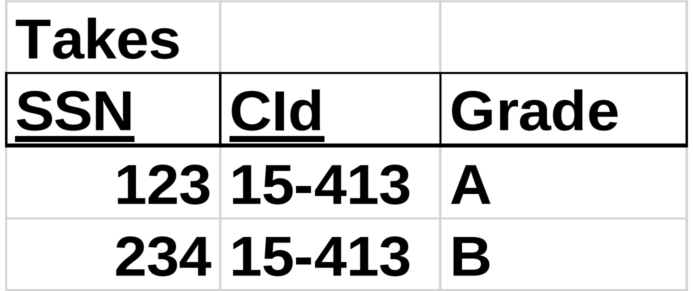
Our Mini-U DB



Students		
SSN	Name	Addr
123	smith	main str
234	jones	QF ave



Classes		
CId	CName	Units
15-413	s.e.	2
15-412	o.s.	2



Takes		
SSN	CId	Grade
123	15-413	A
234	15-413	B

Insertions, Deletions and Updates

Insertions

```
insert into Students(ssn, name, address)
values (123, ‘smith’, ‘main’)
```

OR…

```
insert into Students
values (123, ‘smith’, ‘main’)
```

Deletions

- Delete the record of ‘smith’

```
delete from Students
where name=‘smith’
```

Be careful - it deletes ALL the ‘smith’s!

Updates

- Update the grade to ‘A’ for ssn=123 and course 15-415

```
update takes
set grade=‘A’
where ssn = 123 and cid= ‘15-415’
```

Integrity Constraints- A Review

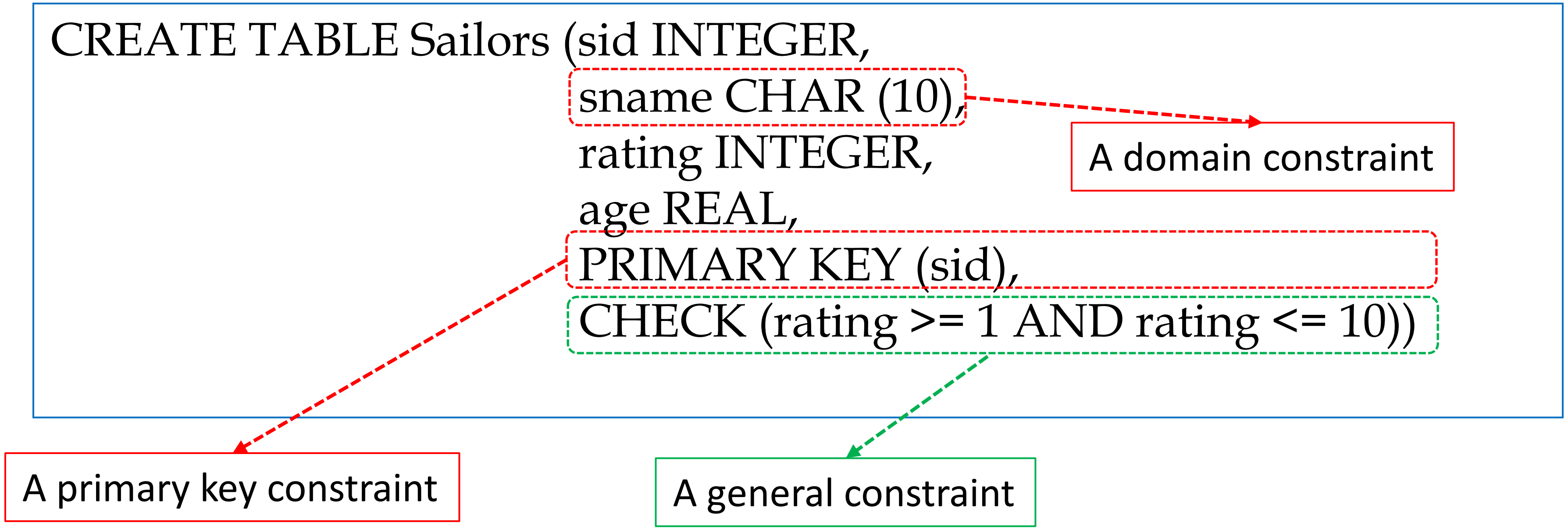
- An Integrity Constraint (IC) describes conditions that every *legal instance* of a relation must satisfy

- Inserts/deletes/updates that violate IC’s are disallowed

- ICs can be used to:
  - Ensure application semantics (e.g., *sid* is a key)
  - Prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 20)

General Constraints Over a Single Table

- Complex constraints over a single table can be defined using **CHECK** *conditional-expression*



```
CREATE TABLE Sailors (sid INTEGER,
                      sname CHAR (10),
                      rating INTEGER,
                      age REAL,
                      PRIMARY KEY (sid),
                      CHECK (rating >= 1 AND rating <= 10))
```

A domain constraint

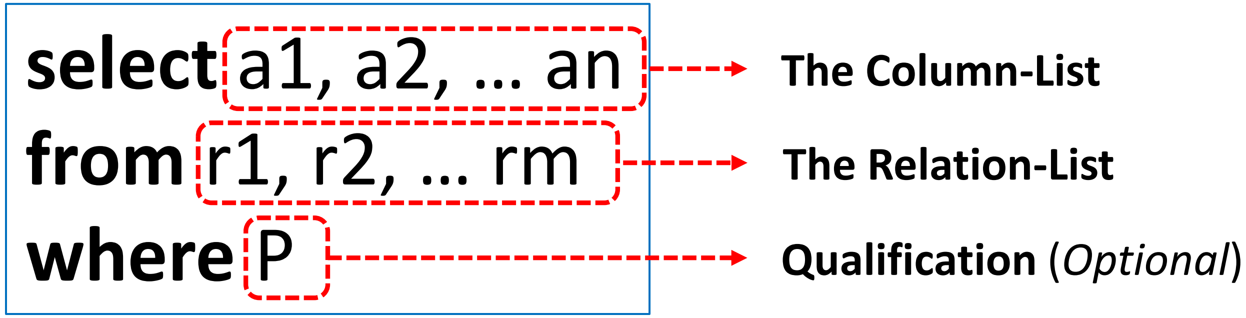
A primary key constraint

A general constraint

Basic SQL Queries

Basic SQL Queries

- The basic form of an SQL query is as follows:



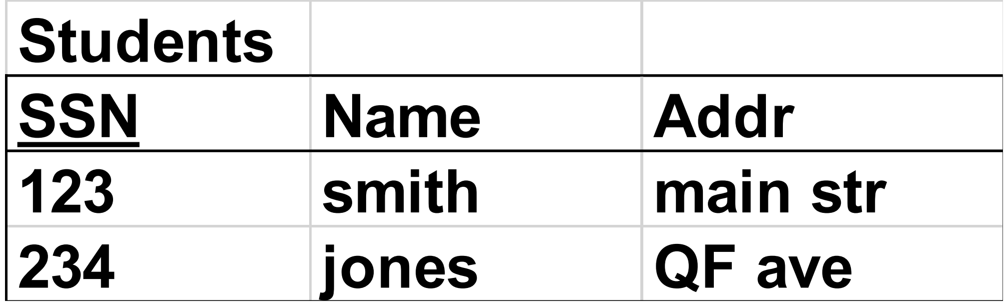
```
select a1, a2, … an
from r1, r2, … rm
where P
```

The Column-List

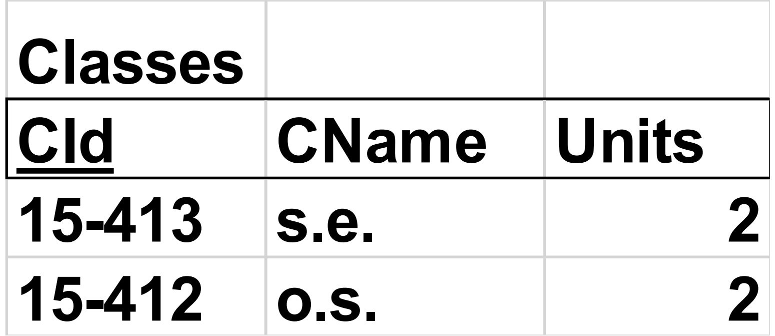
The Relation-List

Qualification (*Optional*)

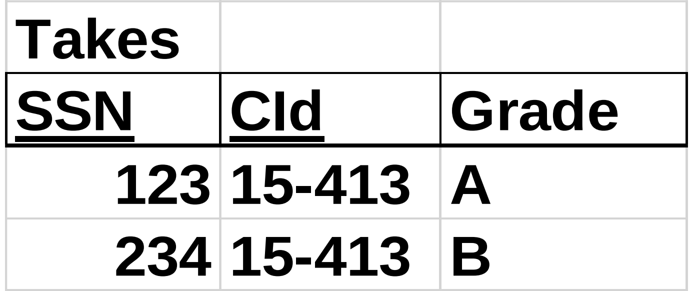
Our Mini-U DB



Students		
SSN	Name	Addr
123	smith	main str
234	jones	QF ave



Classes		
CId	CName	Units
15-413	s.e.	2
15-412	o.s.	2



Takes		
SSN	CId	Grade
123	15-413	A
234	15-413	B

The WHERE Clause

- Find the ssn(s) of everybody called “smith”

Students		
SSN	Name	Addr
123	smith	main str
234	jones	QF ave

```
select ssn
from students
where name=‘smith’
```

The WHERE Clause

- Find ssn(s) of all “smith”s on “main”

Students		
<u>SSN</u>	**Name**	**Addr**
123	**smith**	**main str**
234	**jones**	**QF ave**

```
select ssn
from students
where addr=‘main’ and
   name = ‘smith’
```

The WHERE Clause

- Boolean operators (**and, or, not**)
- Comparison operators (<, ≤, >, ≥, =, ≠)
- And more…

What About Strings?

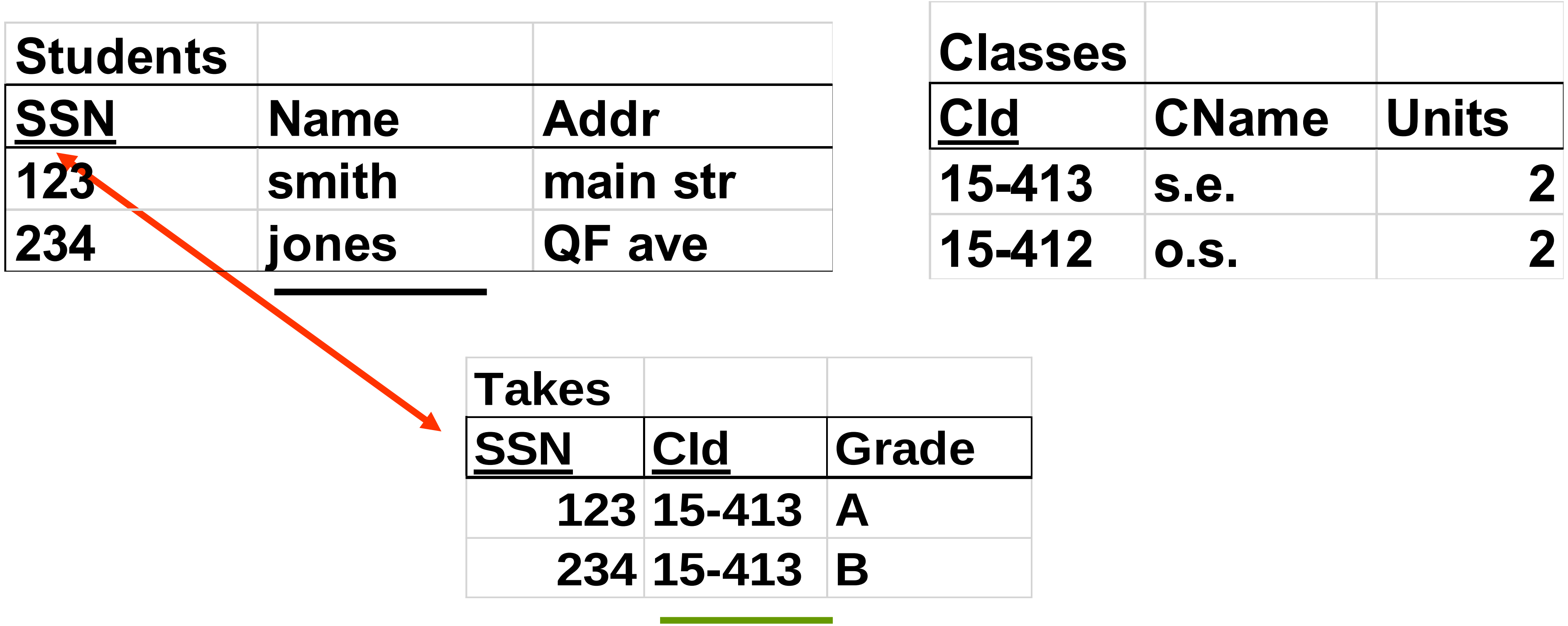
- Find student ssn(s) who live on “main” (st or str or street – i.e., “main st” or “main str” or “main street”)

```
select ssn
from students
where addr like ‘main%’
```

%: Variable-length do not care (i.e., stands for 0 or more arbitrary characters)
_: Single-character do not care (i.e., stands for any 1 character)

The FROM Clause

- Find the names of students taking 15-415



Students		
SSN	Name	Addr
123	smith	main str
234	jones	QF ave

Classes		
CId	CName	Units
15-413	s.e.	2
15-412	o.s.	2

Takes		
SSN	CId	Grade
123	15-413	A
234	15-413	B

The FROM Clause

- Find the names of students taking 15-415

```
select Name
from STUDENTS, TAKES
where   ???
```

The FROM Clause

- Find the names of students taking 15-415

```
select Name
from STUDENTS, TAKES
where   STUDENTS.ssn = TAKES.ssn
        and TAKES.cid = ‘15-415’
```

Renaming: Tuple Variables

- Find the names of students taking 15-415

```
select Name
from STUDENTS as S, TAKES as T
where   S.ssn = T.ssn
         and T.cid = “15-415”
```

Optional!

Renaming: Self-Joins

- Find names and increments for the ratings of persons who have sailed two different boats on the same day

Sailors

Sid	Sname	Rating	age
22	Dustin	7	45.0
29	Brutus	1	33.0

Sailor ID Boat ID **Reserves**

Sid	Bid	Day
22	101	10/10/2013
22	102	10/10/2013

Renaming: Self-Joins

- Find names and increments for the ratings of persons who have sailed two different boats on the same day

Sailors			
Sid	Sname	Rating	age
22	Dustin	7	45.0
29	Brutus	1	33.0

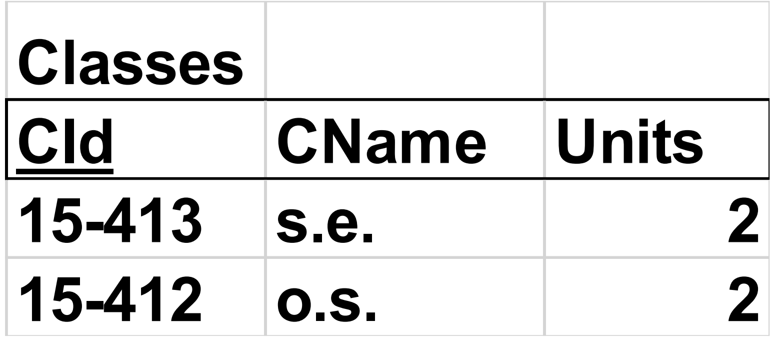
Reserves		
Sid	Bid	Day
22	101	10/10/2020
22	102	10/10/2020

```
select S.sname, S.rating+1 as rating
from Sailors S, Reserves R1, Reserves R2
where S.sid = R1.sid and S.sid = R2.sid
  and  R1.day = R2.day and R1.bid != R2.bid
```

R1 and R2 are different table aliases for the same table.

Renaming: Theta Joins

- Find course names with more units than 15-415



Classes		
CId	CName	Units
15-413	s.e.	2
15-412	o.s.	2

```
select c1.cname
from class as c1, class as c2
where c1.units > c2.units
   and  c2.cid = ‘15-415’
```

Set Operations

Set Operations

- Find ssn(s) of students taking both 15-415 and 15-413

Takes		
SSN	CId	Grade
123	15-413	A
234	15-413	B

~~**select** ssn~~
~~**from** takes~~
~~**where** cid=‘15-415’ **and**~~
~~cid=‘15-413’~~

Set Operations

- Find ssn(s) of students taking both 15-415 and 15-413

Takes		
SSN	CId	Grade
123	15-413	A
234	15-413	B

```
(select ssn from takes where cid=“15-415” )
intersect
(select ssn from takes where cid=“15-413” )
```

Other operations: **union , except**

Set Operations

- Find ssn(s) of students taking 15-415 or 15-413

Takes		
SSN	CId	Grade
123	15-413	A
234	15-413	B

```
(select ssn from takes where cid=“15-415” )
union
(select ssn from takes where cid=“15-413” )
```

Set Operations

- Find ssn(s) of students taking 15-415 <u>but not</u> 15-413

Takes		
<u>SSN</u>	<u>CId</u>	Grade
123	15-413	A
234	15-413	B

```
(select ssn from takes where cid="15-415" )
except
(select ssn from takes where cid="15-413" )
```

SQL Summary

```
INSERT INTO Users (name, email) VALUES ('Kristin', 'kf@umich.edu')

DELETE FROM Users WHERE email='ted@umich.edu'

UPDATE Users SET name="Charles" WHERE email='csev@umich.edu'

SELECT * FROM Users

SELECT * FROM Users WHERE email='csev@umich.edu'

SELECT * FROM Users ORDER BY email
```

See more examples at py4e-ch15-handout.txt

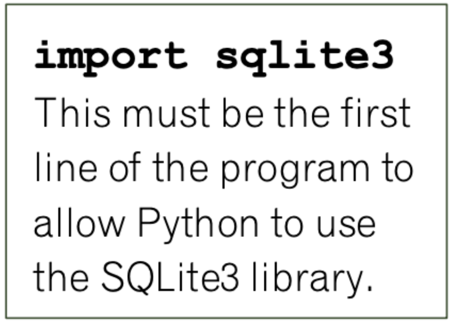
Interacting with SQLite from Python

Database API

- The sqlite3 tool is primarily useful for extracting data from SQLite databases from a script, or quick exploration and interaction with a SQLite database.

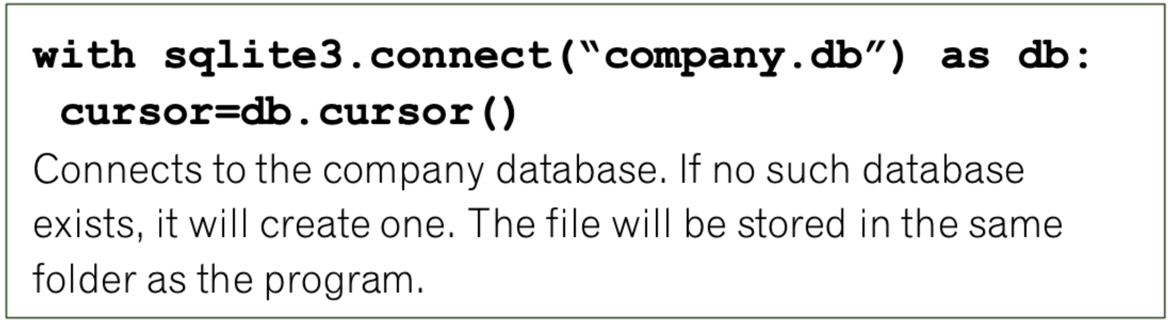
- But for more involved tasks such as loading numerous records into a database or executing complex queries as part of data analysis, it’s often preferable to interact with a SQLite database through an API.
- APIs allow you to interact with a database through an interface in your language of choice. We will take a quick look at Python’s excellent API.

Python module - sqlite3



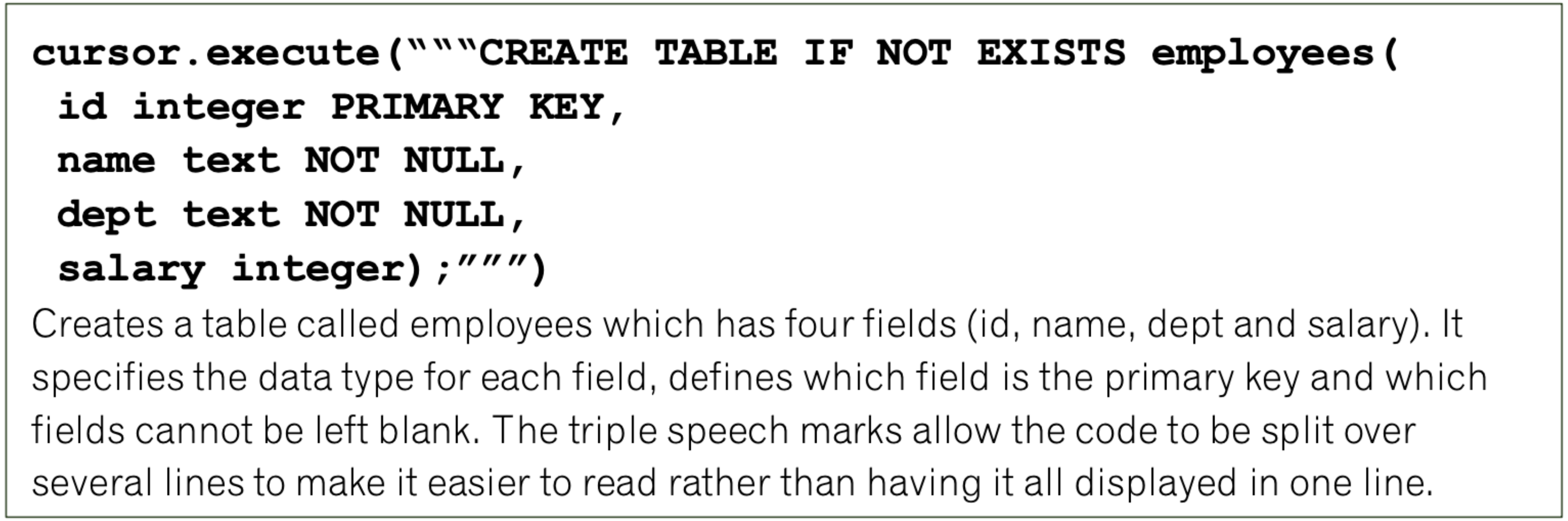
```
import sqlite3
```

This must be the first line of the program to allow Python to use the SQLite3 library.



```
with sqlite3.connect(“company.db”) as db:
 cursor=db.cursor()
```

Connects to the company database. If no such database exists, it will create one. The file will be stored in the same folder as the program.

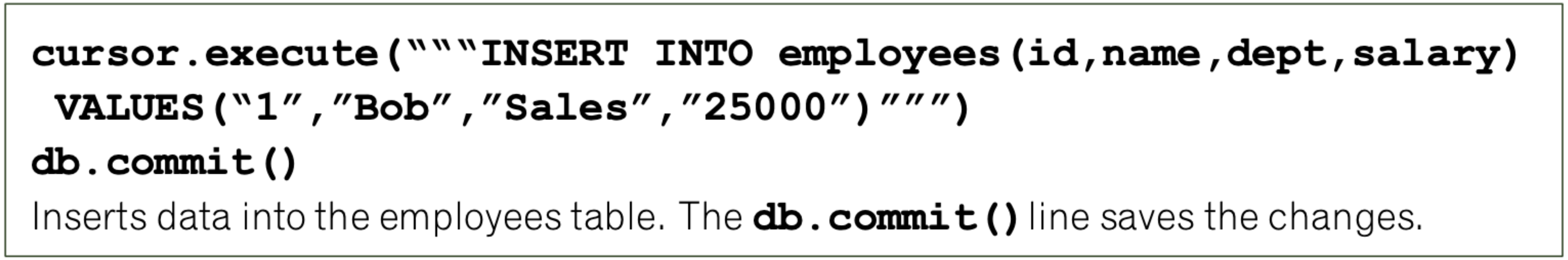


```
cursor.execute(“““CREATE TABLE IF NOT EXISTS employees(
 id integer PRIMARY KEY,
 name text NOT NULL,
 dept text NOT NULL,
 salary integer);”””)
```

Creates a table called employees which has four fields (id, name, dept and salary). It specifies the data type for each field, defines which field is the primary key and which fields cannot be left blank. The triple speech marks allow the code to be split over several lines to make it easier to read rather than having it all displayed in one line.

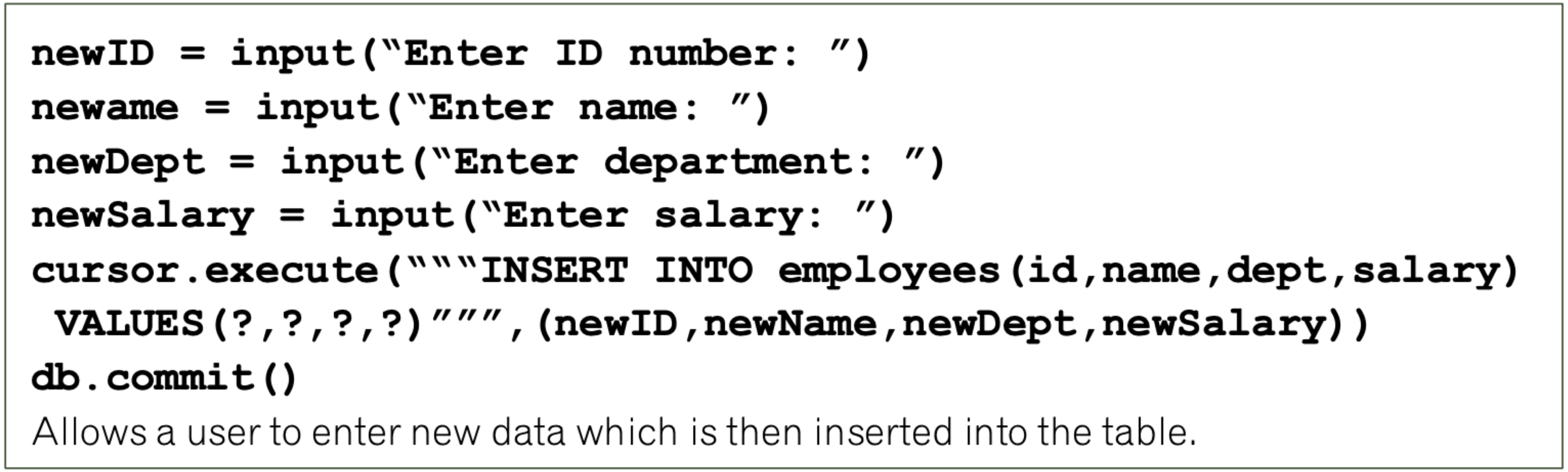
Python by Example, Learning to Program in 150 Challenges, Nichola Lacey 2019

db.cursor()



```
cursor.execute(“““INSERT INTO employees(id,name,dept,salary)
 VALUES(“1”,”Bob”,”Sales”,”25000”)”””)
db.commit()
```

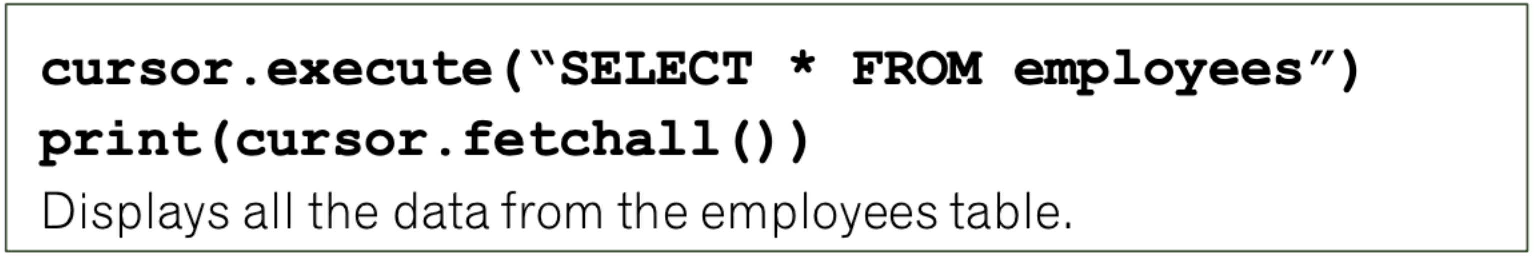
Inserts data into the employees table. The **db.commit()** line saves the changes.



```
newID = input(“Enter ID number: ”)
newame = input(“Enter name: ”)
newDept = input(“Enter department: ”)
newSalary = input(“Enter salary: ”)
cursor.execute(“““INSERT INTO employees(id,name,dept,salary)
 VALUES(?,?,?,?)”””,(newID,newName,newDept,newSalary))
db.commit()
```

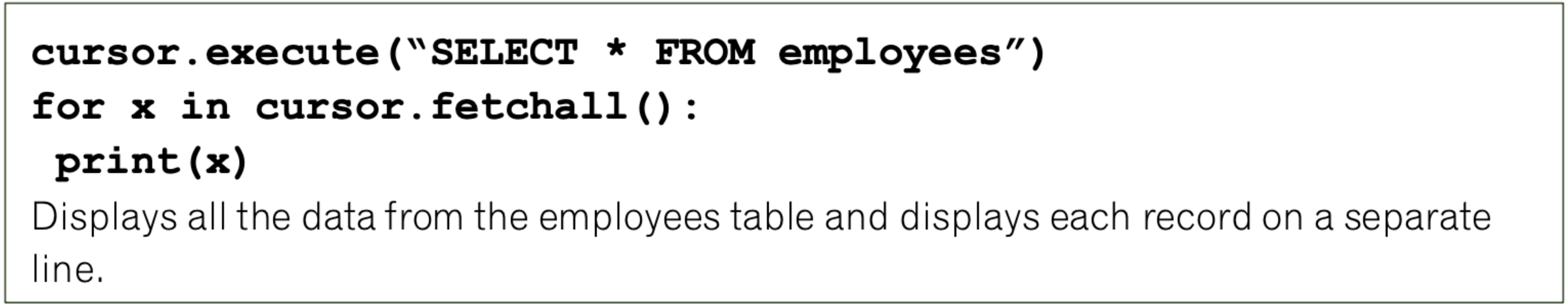
Allows a user to enter new data which is then inserted into the table.

Python by Example, Learning to Program in 150 Challenges, Nichola Lacey 2019



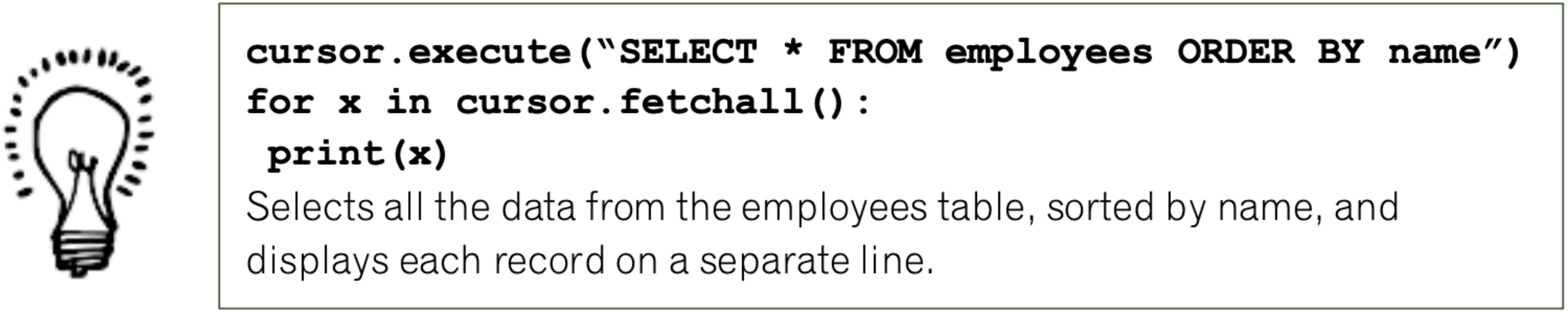
```
cursor.execute(“SELECT * FROM employees”)
print(cursor.fetchall())
```

Displays all the data from the employees table.



```
cursor.execute(“SELECT * FROM employees”)
for x in cursor.fetchall():
 print(x)
```

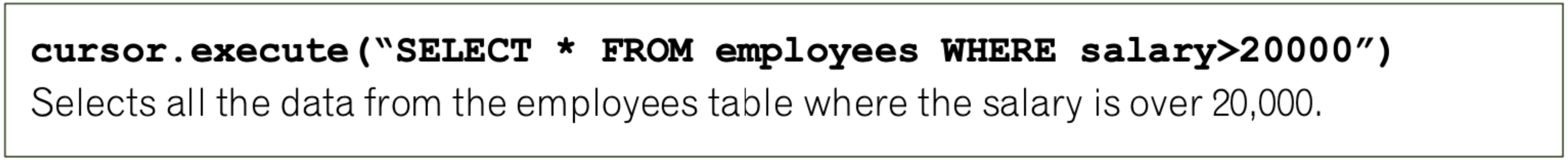
Displays all the data from the employees table and displays each record on a separate line.



```
cursor.execute(“SELECT * FROM employees ORDER BY name”)
for x in cursor.fetchall():
 print(x)
```

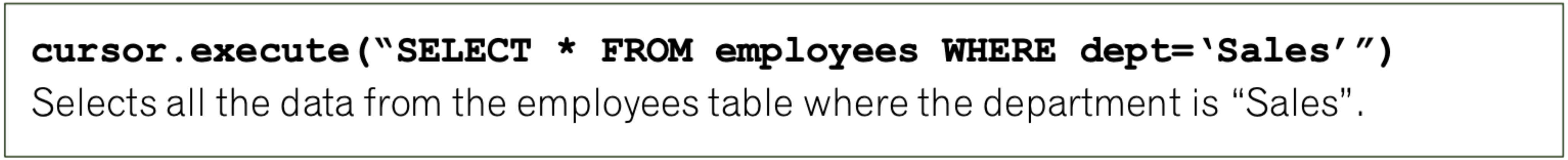
Selects all the data from the employees table, sorted by name, and displays each record on a separate line.

Python by Example, Learning to Program in 150 Challenges, Nichola Lacey 2019



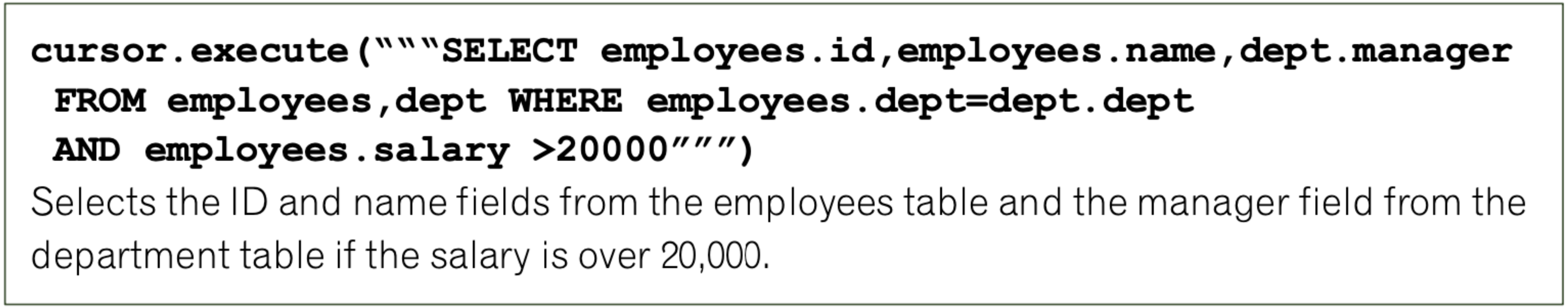
```
cursor.execute(“SELECT * FROM employees WHERE salary>20000”)
```

Selects all the data from the employees table where the salary is over 20,000.



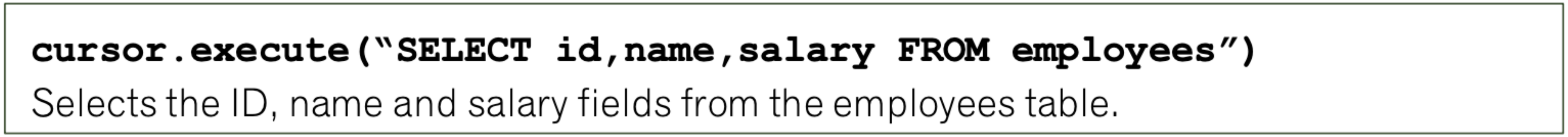
```
cursor.execute(“SELECT * FROM employees WHERE dept=‘Sales’”)
```

Selects all the data from the employees table where the department is “Sales”.



```
cursor.execute(“““SELECT employees.id,employees.name,dept.manager
 FROM employees,dept WHERE employees.dept=dept.dept
 AND employees.salary >20000”””)
```

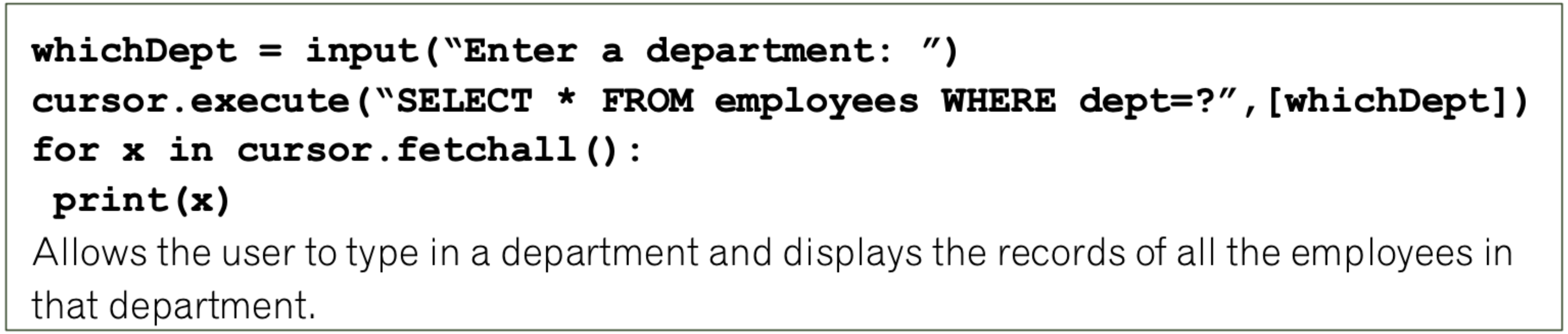
Selects the ID and name fields from the employees table and the manager field from the department table if the salary is over 20,000.



```
cursor.execute(“SELECT id,name,salary FROM employees”)
```

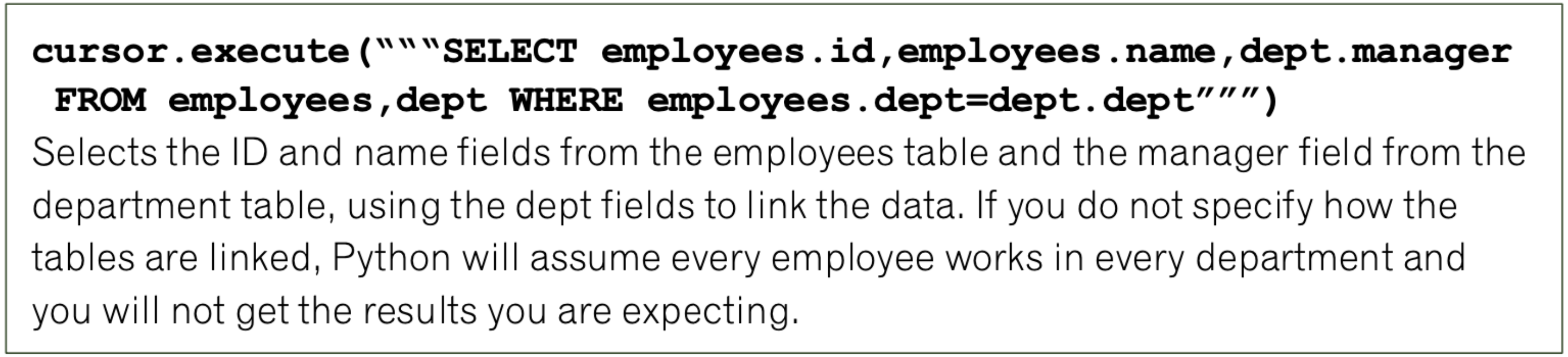
Selects the ID, name and salary fields from the employees table.

Python by Example, Learning to Program in 150 Challenges, Nichola Lacey 2019



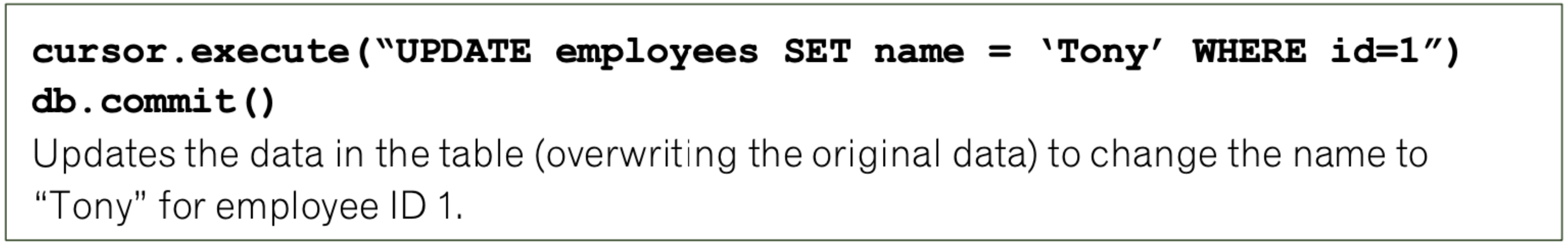
```
whichDept = input(“Enter a department: ”)
cursor.execute(“SELECT * FROM employees WHERE dept=?”,[whichDept])
for x in cursor.fetchall():
 print(x)
```

Allows the user to type in a department and displays the records of all the employees in that department.



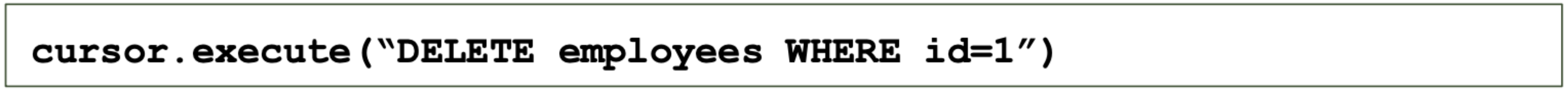
```
cursor.execute(“““SELECT employees.id,employees.name,dept.manager
 FROM employees,dept WHERE employees.dept=dept.dept”””)
```

Selects the ID and name fields from the employees table and the manager field from the department table, using the dept fields to link the data. If you do not specify how the tables are linked, Python will assume every employee works in every department and you will not get the results you are expecting.



```
cursor.execute(“UPDATE employees SET name = ‘Tony’ WHERE id=1”)
db.commit()
```

Updates the data in the table (overwriting the original data) to change the name to “Tony” for employee ID 1.

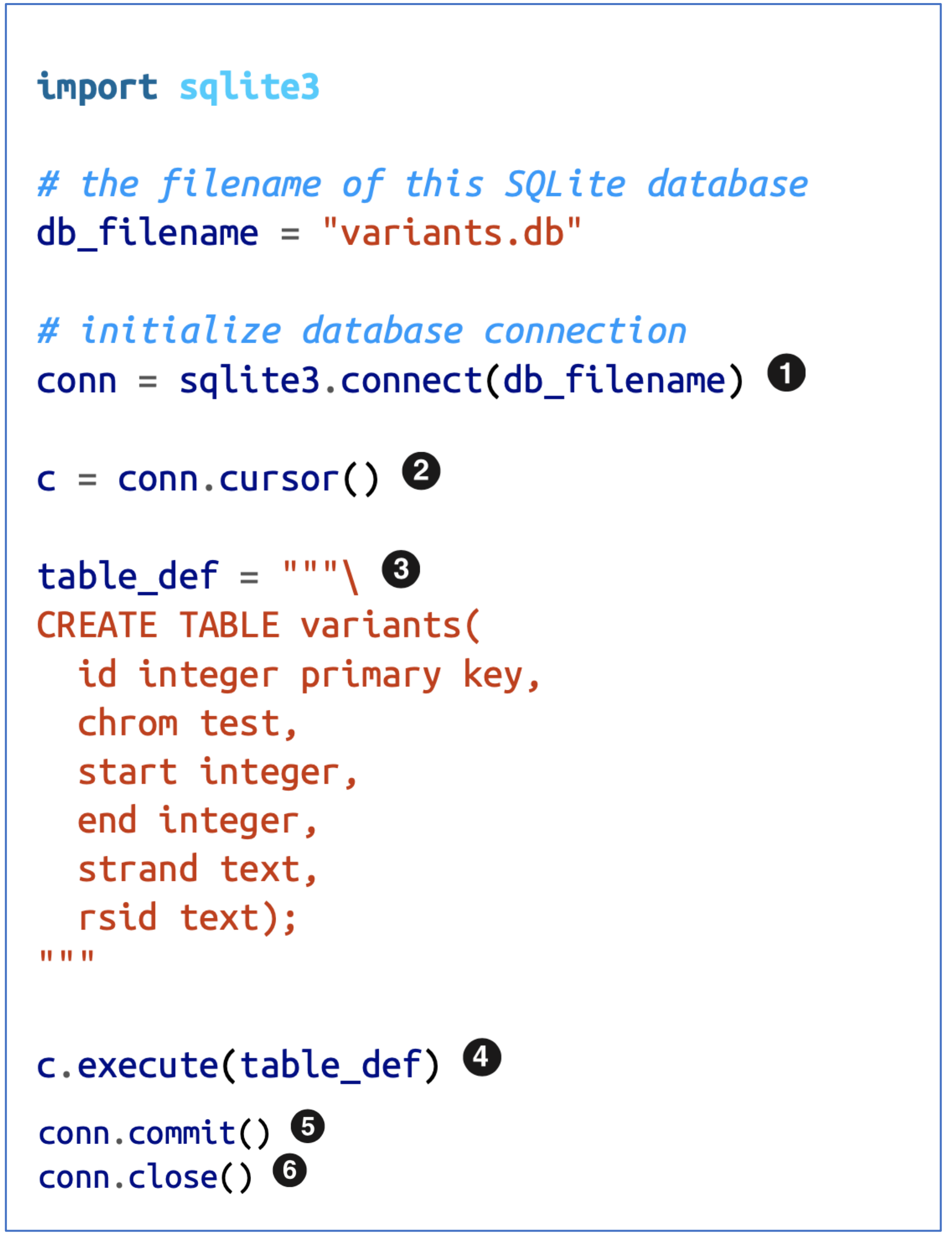


```
cursor.execute(“DELETE employees WHERE id=1”)
```

Python by Example, Learning to Program in 150 Challenges, Nichola Lacey 2019

Connecting to SQLite databases and creating tables from Python

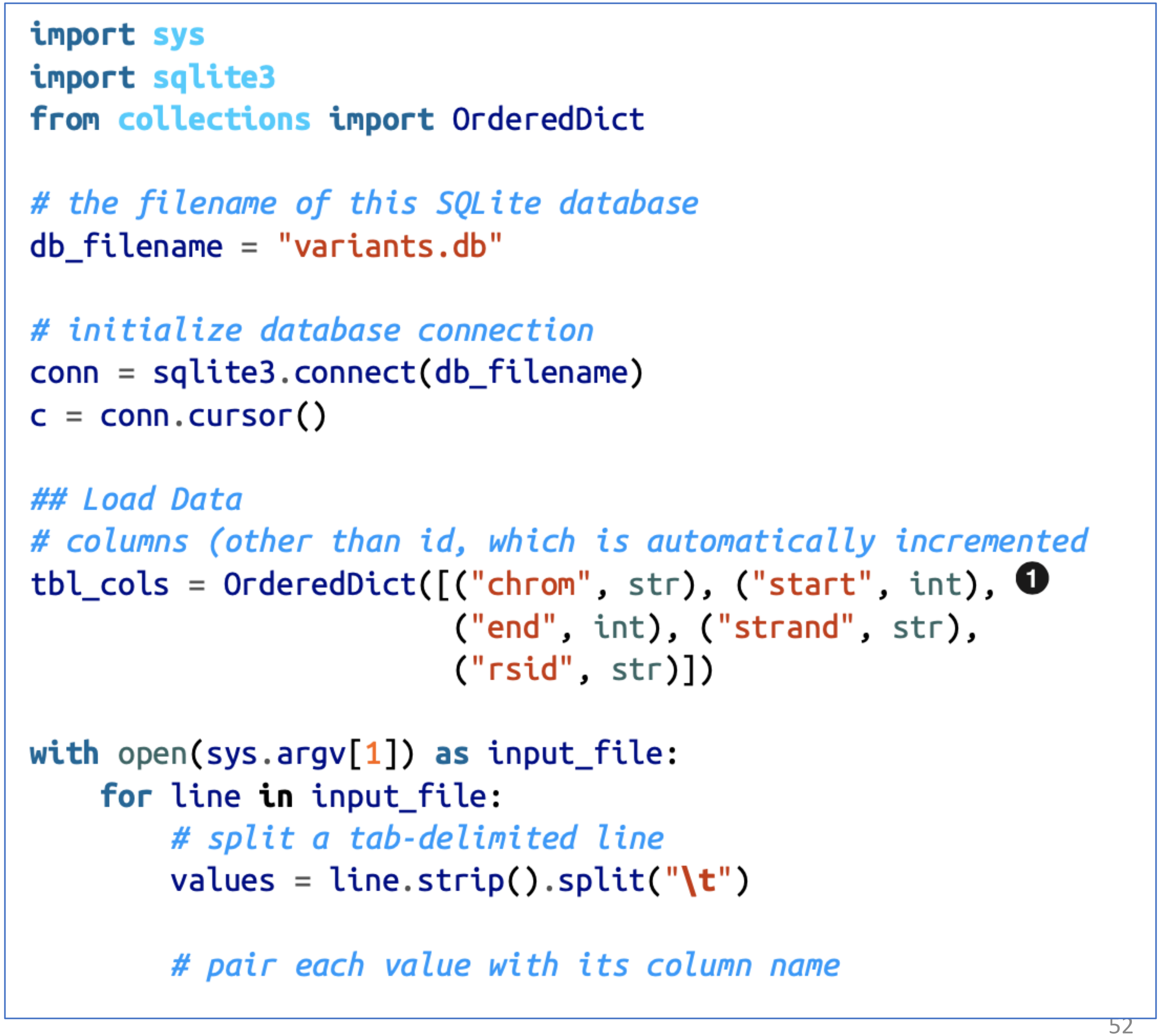
create_table.py



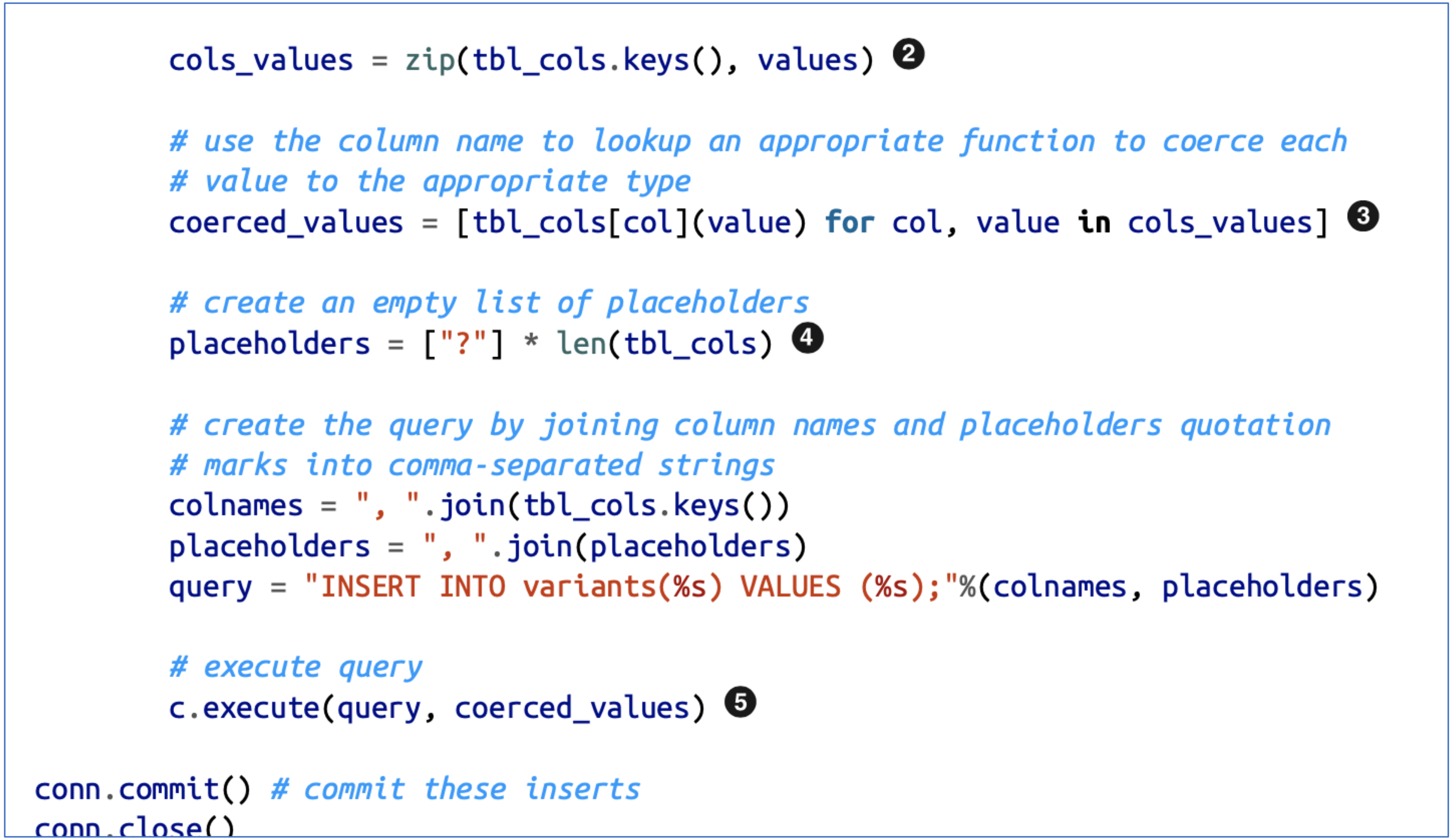
```
import sqlite3

# the filename of this SQLite database
db_filename = "variants.db"

# initialize database connection
conn = sqlite3.connect(db_filename) ❶

c = conn.cursor() ❷

table_def = """\ ❸
CREATE TABLE variants(
  id integer primary key,
  chrom test,
  start integer,
  end integer,
  strand text,
  rsid text);
"""

c.execute(table_def) ❹

conn.commit() ❺
conn.close() ❻
```

Loading data into a table from Python

load_variants.py



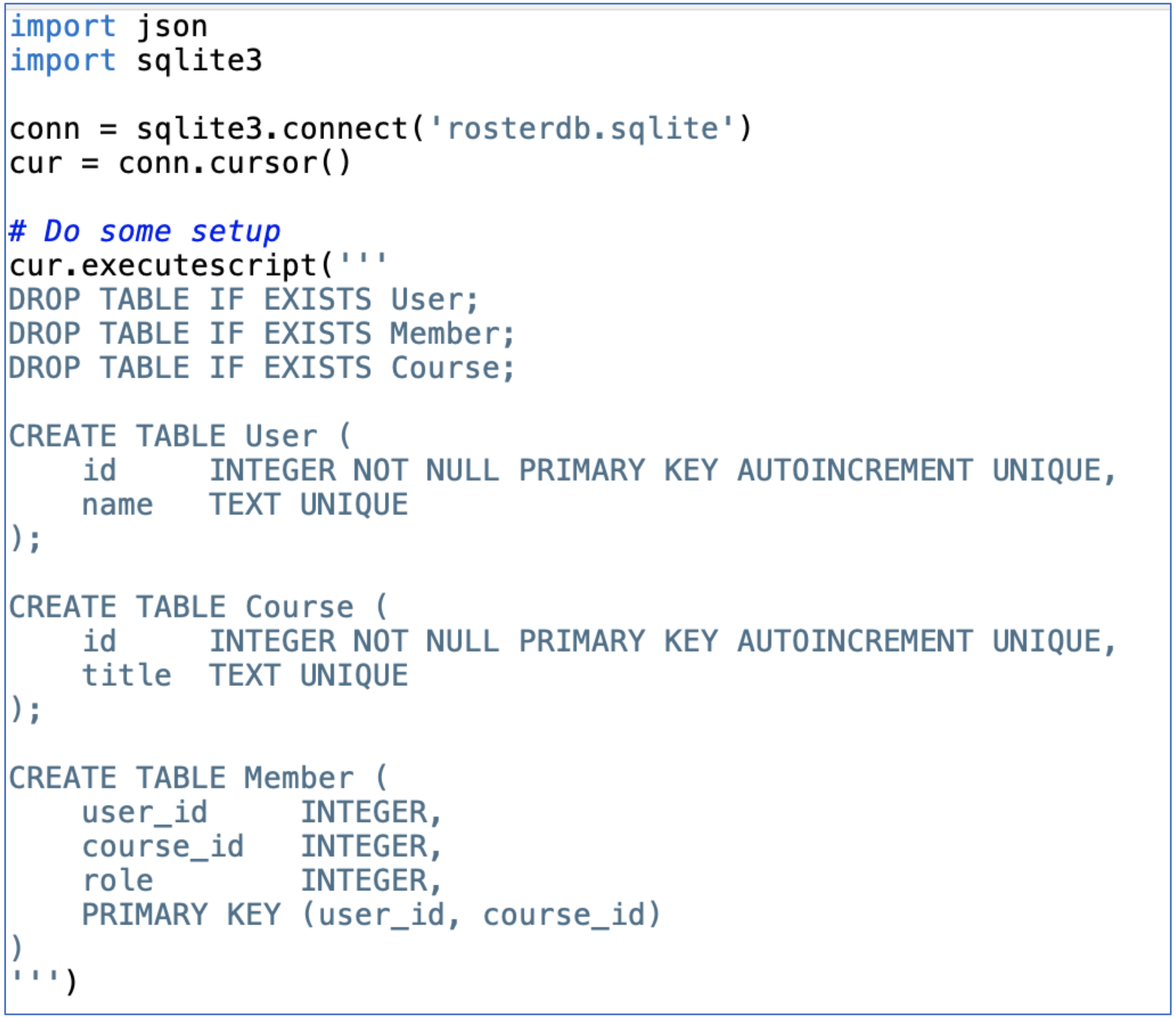
```
import sys
import sqlite3
from collections import OrderedDict

# the filename of this SQLite database
db_filename = "variants.db"

# initialize database connection
conn = sqlite3.connect(db_filename)
c = conn.cursor()

## Load Data
# columns (other than id, which is automatically incremented
tbl_cols = OrderedDict([("chrom", str), ("start", int), ❶
                        ("end", int), ("strand", str),
                        ("rsid", str)])

with open(sys.argv[1]) as input_file:
    for line in input_file:
        # split a tab-delimited line
        values = line.strip().split("\t")

        # pair each value with its column name
```



```
        cols_values = zip(tbl_cols.keys(), values) ❷

        # use the column name to lookup an appropriate function to coerce each
        # value to the appropriate type
        coerced_values = [tbl_cols[col](value) for col, value in cols_values] ❸

        # create an empty list of placeholders
        placeholders = ["?"] * len(tbl_cols) ❹

        # create the query by joining column names and placeholders quotation
        # marks into comma-separated strings
        colnames = ", ".join(tbl_cols.keys())
        placeholders = ", ".join(placeholders)
        query = "INSERT INTO variants(%s) VALUES (%s);"%(colnames, placeholders)

        # execute query
        c.execute(query, coerced_values) ❺

conn.commit() # commit these inserts
conn.close()
```

Loading *.json file into a table from Python

roster.py



```
import json
import sqlite3

conn = sqlite3.connect('rosterdb.sqlite')
cur = conn.cursor()

# Do some setup
cur.executescript('''
DROP TABLE IF EXISTS User;
DROP TABLE IF EXISTS Member;
DROP TABLE IF EXISTS Course;

CREATE TABLE User (
    id     INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
    name   TEXT UNIQUE
);

CREATE TABLE Course (
    id     INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
    title  TEXT UNIQUE
);

CREATE TABLE Member (
    user_id     INTEGER,
    course_id   INTEGER,
    role        INTEGER,
    PRIMARY KEY (user_id, course_id)
)
''')
```

```
fname = input('Enter file name: ')
if len(fname) < 1:
    fname = 'roster_data_sample.json'

# [
#   [ "Charley", "si110", 1 ],
#   [ "Mea", "si110", 0 ],

str_data = open(fname).read()
json_data = json.loads(str_data)

for entry in json_data:

    name = entry[0]
    title = entry[1]

    print((name, title))

    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()
```

Work with the Cursor object

```
>>> import sqlite3
>>> conn = sqlite3.connect("variants.db")
>>> c = conn.cursor()
>>> statement = """\
... SELECT chrom, start, end FROM variants WHERE rsid IN ('rs12255372', 'rs333')
... """

>>> c.execute(statement)
<sqlite3.Cursor object at 0x10e249f80>
>>> c.fetchone()
(u'chr10', 114808901, 114808902)
>>> c.fetchone()
(u'chr3', 46414946, 46414978)
>>> c.fetchone() # nothing left
>>>
```

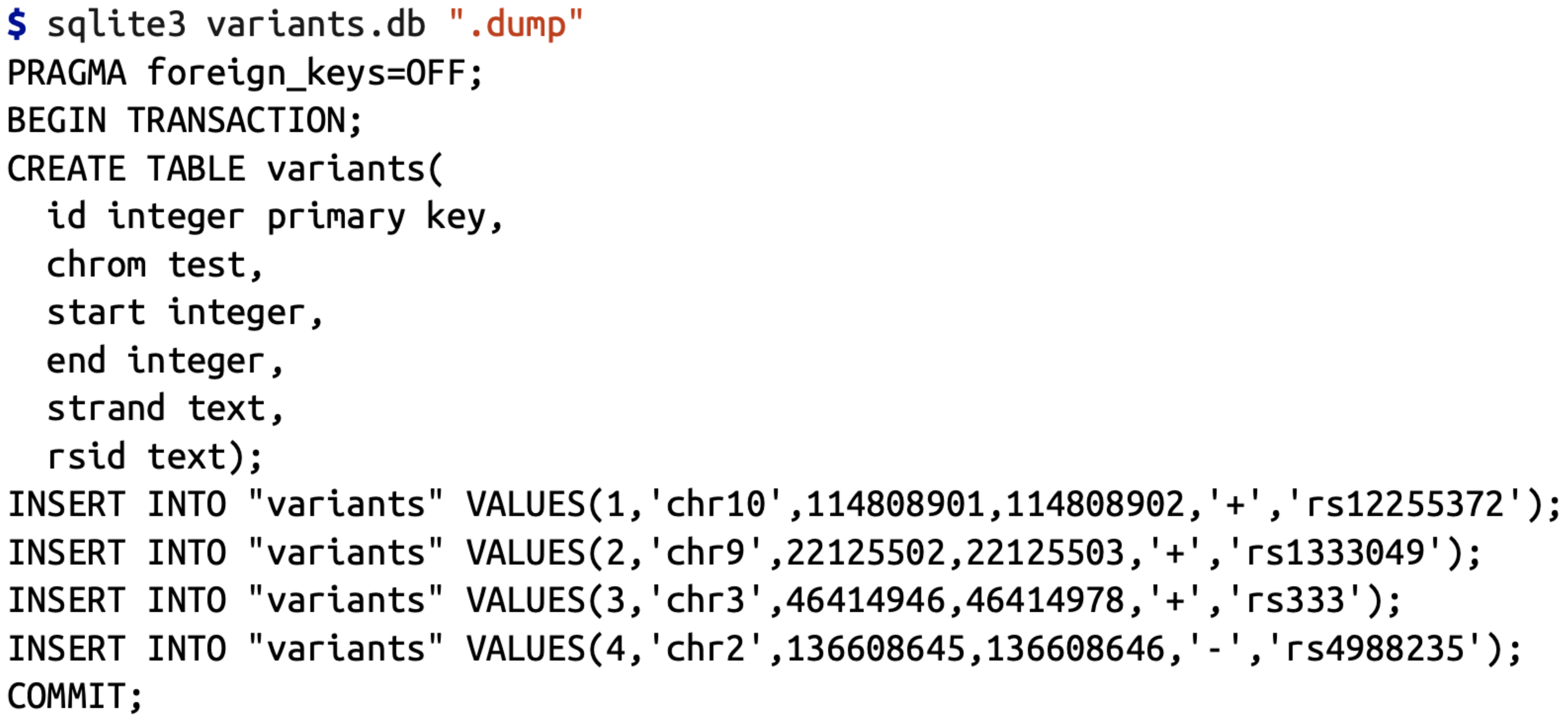
Dumping Databases

Dumping Databases

- useful to back up and duplicate databases
- also useful in sharing databases
    - In SQLite it’s easier to simply share the database file though
    - but this isn’t possible with other database engines like MySQL and PostgreSQL

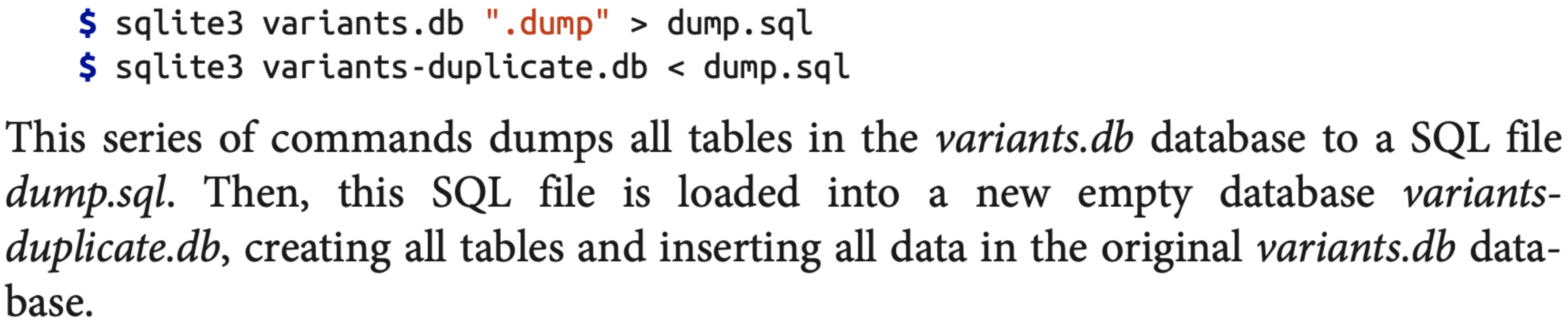
Bioinformatics Data Skills, O'Reilly 2015

Dumping database using SQLite



```
$ sqlite3 variants.db ".dump"
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE variants(
  id integer primary key,
  chrom test,
  start integer,
  end integer,
  strand text,
  rsid text);
INSERT INTO "variants" VALUES(1,'chr10',114808901,114808902,'+','rs12255372');
INSERT INTO "variants" VALUES(2,'chr9',22125502,22125503,'+','rs1333049');
INSERT INTO "variants" VALUES(3,'chr3',46414946,46414978,'+','rs333');
INSERT INTO "variants" VALUES(4,'chr2',136608645,136608646,'-','rs4988235');
COMMIT;
```

Use database dumps to create databases



```
$ sqlite3 variants.db ".dump" > dump.sql
$ sqlite3 variants-duplicate.db < dump.sql
```

This series of commands dumps all tables in the *variants.db* database to a SQL file *dump.sql*. Then, this SQL file is loaded into a new empty database *variants-duplicate.db*, creating all tables and inserting all data in the original *variants.db* database.

Recommended readings

- See materials in the Blackboard.