DATA 520
Lecture 22

Database Programming

SQL and SQLite

Projects

**** Please talk to me before deciding on a project! ****

Presentations November 29, and December 5 and 7

Max of 15 minutes
Introduction:
What problem is the program solving, or what is it doing?
Materials and Methods: Code sources, modifications, functions
Demonstration
Questions

Justin has volunteered for Nov 29

```
# random numbers
random.sample(list(range(1,14)),k=13)
```

[13, 7, 8, 5, 4, 10, 9, 1, 2, 6, 11, 12, 3]

Team?	No	Name
b	**1**	Atluri, Akhilesh
	2	Beezub, Heidi L.
	3	Dubey, Shraddha
	4	Fisher, William J.
a	**5**	Gonzalez, Tiffany M.
	6	Innes, Andrew J.
	7	Le Grange, Stephanie
	8	Minichelli, Judy L.
	9	Minsk, Justin D.
a	**10**	Moncada, Dayana J.
	11	Richardson, Ron
a	**12**	Staudt, Kimberly J.
b	**13**	Varghese, Jerrin J.

Presentation Schedule and Report

Nov 29	Presentation
Wednesday	Justin
Dec 4	
Monday	Jerrin and Akhi
	Stephanie
	Judy
	Tiffany,Dayana,Kim
Dec 6	
Wednesday	Bill
	Heidi
	Andrew
	Ron

After the presentation:

Send me a pdf copy of your presentation

Short report due Dec 15 at 11:59:

2 page **maximum** explanatory text;

- then add all code borrowed, modifications (comments), tests run

FINAL EXAM
Wednesday December 13
3:30 - 5:30

Database Basics

Data are arranged in rows and columns (records and fields) in **TABLES**

A collection of tables is a **DATABASE**

Database fields allow only certain types of data (Text/Integer/Numeric/Boolean/Date)

Database fields have **display rules** (39375 = 10/20/2007 in US date format)

Data can be extracted using various programs, often **SQL**

(Structured Query Language)

Data can be exported in various formats for use in other programs

Comma-delimited probably best export/import format, space-delimited probably the worst.

```
id,gol,xcb,bbh
0301,190,145,140
0401,188,142,139
```

Excel can be useful but it sucks for important data

Excel is NOT a database!

Columns and rows can be moved around easily

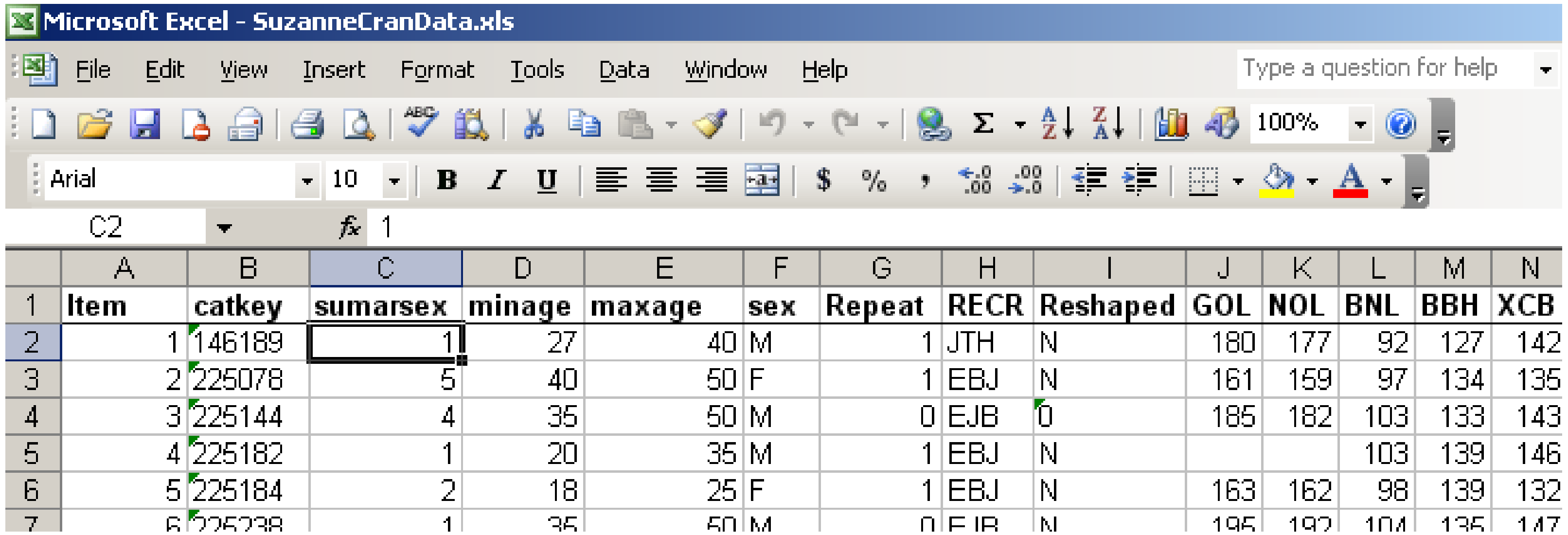
Excel changes data types spontaneously (especially to DATES)

Excel would mangle data types when exporting to a database format

- was often due to a blank value in the first record

Cells can contain formulas rather than values

No memo fields (text > 255 characters)


Microsoft Excel - SuzanneCranData.xls
File Edit View Insert Format Tools Data Window Help
Type a question for help
100%
Arial
10
C2
1

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Item	catkey	sumarsex	minage	maxage	sex	Repeat	RECR	Reshaped	GOL	NOL	BNL	BBH	XCB
2	1	146189	1	27	40	M	1	JTH	N	180	177	92	127	142
3	2	225078	5	40	50	F	1	EBJ	N	161	159	97	134	135
4	3	225144	4	35	50	M	0	EJB	0	185	182	103	133	143
5	4	225182	1	20	35	M	1	EBJ	N			103	139	146
6	5	225184	2	18	25	F	1	EBJ	N	163	162	98	139	132
7	6	225238	1	35	50	M	0	EJB	N	195	192	104	135	147

Database Field Types

Field Type

Character/Text (alphanumeric, up to 255 places)

Date	2.3.2008, 3/2/2008, 20080302
Short integer	-256 to 256
Long integer	+/-999 trillion
Numeric / Single / Double / Float	34.6657
Boolean	False, 0 or True, 1
Varchar(2000)	Text with paragraphs, very long (memo)
BLOB	Binary Large Object (jpeg)

Spaces are bad in field names

In field names: “head ln” → “HeadLn” or “Head_Ln”

In field values (museum number):

"HTH 102" → "HTH102" or "HTH_102"

Use leading zeros if you care about sorting!

HTH102	→	**HTH102**		**HTH004**
HTH4	→	**HTH004**	**sorted:**	**HTH099**
HTH99	→	**HTH099**		**HTH102**

- Same problem with “A00000073-1”

Good Data Practices

Have a “key field”, at least one field with unique values*

Set up identifying text fields as the first ones

Leave out memo fields and erroneous text fields for analysis

Remove spaces in field names **ALWAYS**

Remove spaces in field values if possible

SHORTEN field names to < 8 characters for display, tables, etc.

- or for use with SAS, SYSTAT, R

*A key field is ESPECIALLY important in the relational database model – ie, ways to link records in a table to information in other tables

Databases

A database stores data in tables

Tables store data in records & fields (rows & columns / tuples & attributes)

A database keeps records and fields safe and stable (data AND structure)

Column order does not matter – data can be retrieved in any order (VIEW)

Data order does not matter much – can always be sorted

Databases allow record inserts, deletes, updates (edits)

Databases allow comparisons and JOINS of data tables for greater insight

Database Programs (History)

Lite
dBase (OpenOffice)
Access (Microsoft)
Paradox
SQLite

Medium
mySQL (open source, then Oracle)

Heavy
Adaptive Server Anywhere (Sybase)
Oracle
SQL Server (Microsoft)
mySQL

Database (backend) Criteria

Portable (API and file format)

Decent interface (connections, import, export data, utilities)

Database management

- SQL compliant , create databases, tables, indices, fields, etc.
- transactional (ACID: atomicity, consistency, isolation, durability)

Y/N done fifo safe

Little/no configuration necessary

Low memory requirements

SQLite

SQLite is small code base 500 KB) with API (embedded)

- files are cross-platform
- can use tables up to 140 TB (but 10-50 GB is the practical limit)
- can write database in memory, then to disk, in one file

SQLite most used DB (cellphones, mp3 players, much software)

Limitations:

- no networking (tcp/ip) – only local / on mappable drives
- low concurrency (if many writes from many users to same db)
- can't password-protect files

Create a Database

```
# first, load the library
>>> import sqlite3
```

A database is a "container"

```
# then create a connection to an empty database, will be a system filename
>>> con = sqlite3.connect('population.db')

# an "in-memory" database:
>>> con2 = sqlite3.connect(':memory:')


# then create a cursor – (a handle) an identifier that you are there:

>>> cur = con.cursor()

Now we can do some things like create tables, populate tables, query tables
```

SQL:

CREATE TABLE <tablename>(<fieldname>,<fieldtype> …)

CREATE TABLE PopByRegion(Region **TEXT**, Population **INTEGER**)

SQLite

Kinds of data in SQLite tables

Type	Python Equivalent	mySQL	Use
NULL	NoneType	-	Means “know nothing about it”
INTEGER	int	BIGINT, TINYINT	Integers
REAL	float	DOUBLE,SINGLE	8-byte floating-point numbers
TEXT	str	CHAR, VARCHAR	Strings of characters
BLOB	bytes	BLOB	Binary data

Binary large object = jpeg, mp3, etc.

Create a table, insert records

SQL: create a table

```
CREATE TABLE PopByRegion(Region TEXT, Population INTEGER)
```

In Python:

```
>>> cur.execute('CREATE TABLE PopByRegion(Region TEXT, Population INTEGER)')
<sqlite3.Cursor object at 0x102e3e490>
```

SQL: Insert records into a table:

column order matters, data type matters (str, number), count matters

```
INSERT INTO <TableName> VALUES(<Value>, ...)
# statement is a string, so are string values
# numbers: no quotes unless they are in string form

>>> cur.execute('INSERT INTO PopByRegion VALUES("Central Africa", 330993)')
>>> cur.execute('INSERT INTO PopByRegion VALUES("Southeastern Africa", 743112)')
>>> cur.execute('INSERT INTO PopByRegion VALUES("Japan", 100562)')
<sqlite3.Cursor object at 0x102e3e490>
```

Integers get converted into floats if field type is float

```
# Another way: using question marks – useful in loops
>>> cur.execute('INSERT INTO PopByRegion VALUES (?, ?)', ("Japan", 100562))
```

Insert multiple records

The trick is to use different quotation marks

```
insert into <Tablename>(<Fieldnames>) values (<Values for each record and column>)

insert into PopByRegion (Region,Population) values ('Southeastern Africa', 743112),
('Japan', 100562)


>>> cur.execute('insert into PopByRegion (Region,Population) values ("Central
Africa", 330993), ("Southeastern Africa", 743112), ("Northern Africa", 1037463),
("Southern Asia", 2051941), ("Asia Pacific", 785468), ("Middle East", 687630),
("Eastern Asia", 1362955), ("South America", 593121), ("Eastern Europe", 223427),
("North America", 661157), ("Western Europe", 387933), ("Japan", 100562)' )
```

OR:

```
>>> cur.execute("insert into PopByRegion (Region,Population) values ('Central
Africa', 330993), ('Southeastern Africa', 743112), ('Northern Africa', 1037463),
('Southern Asia', 2051941), ('Asia Pacific', 785468), ('Middle East', 687630),
('Eastern Asia', 1362955), ('South America', 593121), ('Eastern Europe', 223427),
('North America', 661157), ('Western Europe', 387933), ('Japan', 100562)" )
```

Insert multiple records

SQL: inserting multiple records with better formatting

```
>>> sql = """
insert into PopByRegion (Region,Population) values
("Central Africa", 330993),
("Southeastern Africa", 743112),
("Northern Africa", 1037463),
("Southern Asia", 2051941),
("Asia Pacific", 785468),
("Middle East", 687630),
("Eastern Asia", 1362955),
("South America", 593121),
("Eastern Europe", 223427),
("North America", 661157),
("Western Europe", 387933),
("Japan", 100562) """
<sqlite3.Cursor object at 0x02D6DA20>

>>> cur.execute(sql)
```

Commit, query a table

```
# Our inserts have not been saved yet!
# To save changes from table operations, we COMMIT them
con.commit()
```

Now we can query the table

SQL:

SELECT <fieldname> [or *] FROM <tablename>

In Python using SQLite:

```
cur.execute('SELECT Region, Population FROM PopByRegion')
... and nothing seems to happen, but...
```

Get records from a query

But something did happen:

```
# get one tuple / record
>>> cur.fetchone()
('Central Africa', 330993)

>>> cur.fetchall()
[('Southeastern Africa', 743112), ('Japan', 100562), ('Japan', 100562)]
```

- they are tuples inside a list

- not all records, but all since the last cursor operation - similar to reading a file

```
>>> cur.fetchone()
```

- nothing returned. The cursor is at the end of the returned records

Get records from a query

The DB Process: 1. CREATE , INSERT; 2. COMMIT; 3. SELECT; 4. FETCH.

SQLite will accept clear mistakes

```
# create a bad record
>>> cur.execute('insert into PopByRegion (Population,Region) values ("Mistake","Mistake")' )

>>> cur.execute('SELECT Region, Population FROM PopByRegion')
>>> cur.fetchall()
[('Central Africa', 330993), ('Southeastern Africa', 743112), ('Japan', 100562), ...
, ('Japan', 100562), ('Central Africa', 330993), ('Mistake', 'Mistake')]

>>> cur.execute('SELECT min(Population) FROM PopByRegion')
>>> cur.fetchall()
[(100562,)]

>>> cur.execute('SELECT Region, max(Population) FROM PopByRegion')
<sqlite3.Cursor object at 0x016FD9E0>
>>> cur.fetchall()
[('Mistake','Mistake')]
```

Get records from a query

Data returned in the order it was entered (arbitrary)
- but we can sort results

```
SQL:
SELECT * from <tablename> ORDER BY <fieldname> [ASC or DESC]

>>> cur.execute('SELECT Region, Population FROM PopByRegion ORDER BY Region')
>>> cur.fetchall()


[('Southeastern Africa', 743112), ('Northern Africa', 1037463), ('Southern
Asia', 2051941), ('Asia Pacific', 785468), ('Middle East', 687630),
('Eastern Asia', 1362955), ('South America', 593121), ('Eastern Europe',
223427), ('North America', 661157), ('Western Europe', 387933), ('Japan',
100562)]
```

Setting conditions with a query: WHERE

```
SQL: WHERE                                                                       operator
SELECT <Fieldnames> FROM <Tablename> WHERE <Fieldname> [=,>,<...] value
SELECT TotalD from Revenue WHERE LName = 'Jones'
SELECT Region FROM PopByRegion WHERE Population > 1000000
SELECT TotalD FROM Revenue WHERE "Bad Field" > Good_Field

>>> cur.execute('SELECT Region FROM PopByRegion WHERE Population > 1000000 ' )
```

Operator	Description
=	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Editing a record: UPDATE

```
>>> cur.execute('SELECT * FROM PopByRegion WHERE Region = "Japan"')
<sqlite3.Cursor object at 0x102e3e490>
>>> cur.fetchone()
('Japan', 100562)

# SQL: UPDATE PopByRegion SET Population = 100600 WHERE Region = "Japan"
# simple way
>>> cur.execute('UPDATE PopByRegion SET Population = 100600 WHERE Region = "Japan" ')
<sqlite3.Cursor object at 0x016FD9E0>

# get around quotation mark problems: use three in a row, works with other single quotes
>>> cur.execute('''UPDATE PopByRegion SET Population = 100600 WHERE Region = 'Japan' ''')
<sqlite3.Cursor object at 0x016FD9E0>

# Three in a row also work with double quotes
>>> cur.execute('''UPDATE PopByRegion SET Population = 100600 WHERE Region = "Japan" ''')
<sqlite3.Cursor object at 0x102e3e490>


>>> cur.execute('SELECT * FROM PopByRegion WHERE Region = "Japan"')
<sqlite3.Cursor object at 0x102e3e490>
>>> cur.fetchone()
('Japan', 100600)
```

Updates and NULL

```
>>> cur.execute('UPDATE PopByRegion SET Population = NULL WHERE Region = "Mistake" ')
<sqlite3.Cursor object at 0x016FD9E0>

>>> cur.execute('SELECT Region, max(Population) FROM PopByRegion')
<sqlite3.Cursor object at 0x016FD9E0>

>>> cur.fetchall()
[('Southern Asia', 2051941)]

# Let's look at the Mistake record with NULL
cur.execute('SELECT Region, Population FROM PopByRegion where Population = NULL')
<sqlite3.Cursor object at 0x016FD9E0>
>>> cur.fetchall()
[]

# Is it gone? I only changed the Population to NULL ------------- I have to use IS!
>>> cur.execute('SELECT Region, Population FROM PopByRegion where Population IS NULL')
<sqlite3.Cursor object at 0x016FD9E0>
>>> cur.fetchall()
[('Mistake', None)]
```

ALWAYS use `IS NULL` when querying

Deleting a record: DELETE

```
>>> cur.execute('UPDATE PopByRegion SET Population = NULL WHERE Region = "Mistake" ')
<sqlite3.Cursor object at 0x016FD9E0>

>>> cur.execute('SELECT Region, max(Population) FROM PopByRegion')
<sqlite3.Cursor object at 0x016FD9E0>

>>> cur.fetchall()
[('Southern Asia', 2051941)]

# SQL: DELETE FROM PopByRegion WHERE Region = "Japan"
>>> cur.execute('DELETE FROM PopByRegion WHERE Region = "Japan" ')

# delete all records
>>> cur.execute('DELETE FROM PopByRegion' )

# delete table: DROP *****
>>> cur.execute('DROP TABLE PopByRegion' )

# close DB connection
>>> con.close()
```

Homework 17

Gries 17.10, page 361:

Problem 1 (a - j)

AND

Convert the Chinese Zodiac data to a SQLite database and run the program using SQL and the database.

DUE: December 1 at 11:59 pm.