DATA SOCIETY®

The premiere data science training for professionals

"One should look for what is and not what he thinks should be."

- Albert Einstein

Objectives

1. Create a database with MAMP and phpMyAdmin
2. Connect your database to a Python script
3. Run basic SQL queries on a sample data set
4. Use an object relational mapper (ORM) for larger data sets

Outline

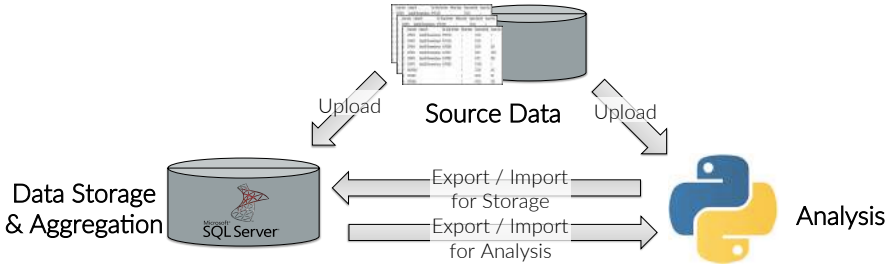
1. Overview of SQL & importing data
2. Connecting to your database in Python
3. Adjusting tables and foreign keys
4. Creating tables in a database
5. Performing basic statements in SQL
6. Performing UNIONS and JOINS in SQL
7. Advanced: using ORMs for larger databases

What is SQL?

- SQL is short for Structured Query Language. It is the standard language used to communicate with most relational databases
  - SQL queries are sent to a database to ask it to perform a specific task with the data it stores
- Microsoft SQL Server is a type of relational database

Goals for Intro to SQL

- Learn to Import/Export, manipulate, combine, and aggregate data sets using MySQL
- These lessons are structured with the intent of using SQL server as an intermediary to store and aggregate data as displayed in the image below



Why use SQL?

Great *Average* *Limited*

	SQL	Access	Excel
Data Size Limits	Best option for analyzing large data sets (over 1 million records)	Handles larger data sets than Excel, but can be limited by memory and space of local computers	Limited to 1,048,576 records for data sets
Manipulating Data	Multiple queries can easily be combined to coerce data from multiple data sets	Querying capabilities similar to SQL with less flexibility and capabilities	Combining data sets can be difficult and prone to manual and formula errors
Analyzing / Reporting Data	Limited built-in analysis functions and lack of built-in reporting and visualization capabilities	Built in form and report capabilities for easy reporting, but more limited set of analysis functions	Many built in analysis functions, visualizations, and formatting for easy modeling and reporting
Speed	Faster Processing	Slower processing	Slower Processing
Compatibility	Compatible with most visualization, business intelligence, and statistical analysis platforms	Compatible with many visualization, business intelligence, and statistical analysis platforms	Compatible with most visualization, business intelligence, and statistical analysis platforms
Quality Control	SQL Queries create repeatable and auditable analysis processes that can be clearly commented	Access Queries also create repeatable processes, but are often less transparent than SQL queries	Excel Analyses are harder to replicate due to manual steps that can be difficult to audit
Learning Curve	Can be easy to learn for people without a programming background	Can be easy to learn for people without a programming background	Easy to learn for any analyst

Outline

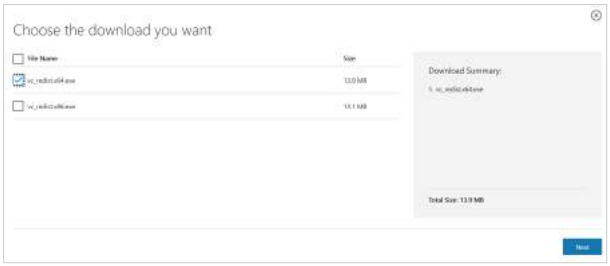
1. Overview of SQL & importing data
2. Connecting to your database in Python
3. Adjusting tables and foreign keys
4. Creating tables in a database
5. Performing basic statements in SQL
6. Performing UNIONS and JOINS in SQL
7. Advanced: using ORMs for larger databases

Download install file

- Go to https://dev.mysql.com/downloads/workbench/
  - Notice the section called "MySQL Workbench Windows Prerequisites." You may need to come back to this page, so keep it open.
  - Go to the bottom of the page and click the Download button.
- You will be asked to create an Oracle Web account in order to proceed. You can click on "No thanks, just start my download."

Check prerequisites

- Double click on the downloaded .msi file to begin installation of MySQL Workbench.
- You may receive a warning that you need either Visual C++ and / or .NET. If so, go back to the MySQL website "Prerequisite" section and download the appropriate requirement(s).
  - If installing Visual C++, choose the x64 version (shown below).
- Once the prerequisites are met, re-launch the Workbench installer.



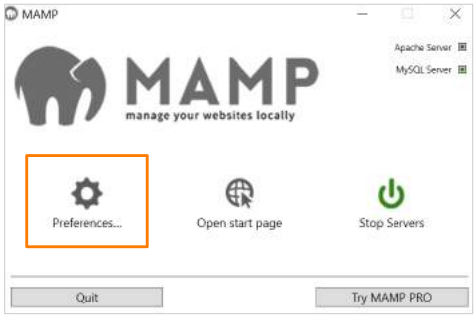
Run installer

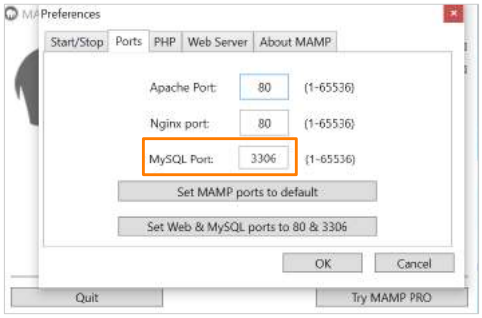
- Click through the installation guide:
  - Install location: Keep default
  - Setup Type: "Complete"
  - Confirmation screen: Click "Install"
- When finished, you can check the box to launch MySQL Workbench, and click Finish.
- MySQL Workbench will open.

Connect to your database server

Now we will set up MySQL Workbench to connect to our database server. Make sure you have completed the MAMP installation steps before attempting this part.

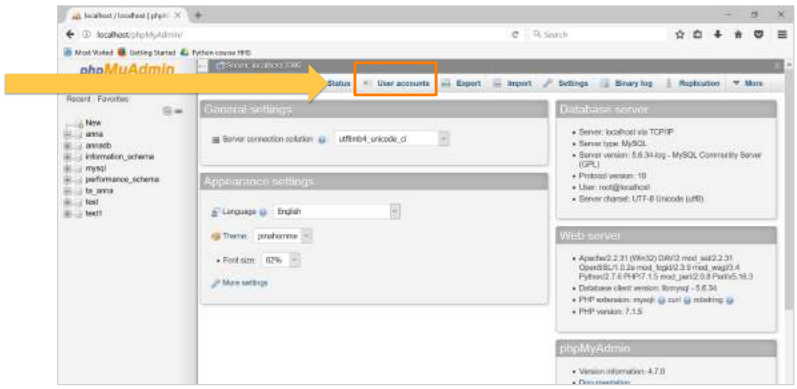
- Start MAMP Servers.
- Go to Preferences > Ports and take note of the MySQL Port number (probably 3306).





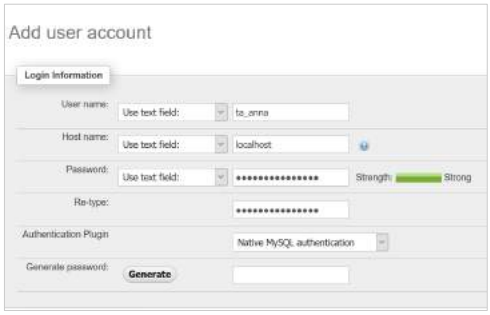
Connect to your database server

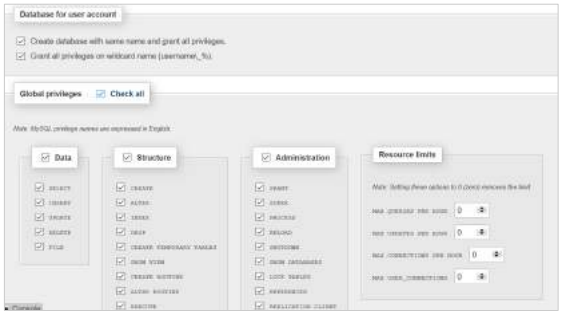
- Open http://localhost/phpMyAdmin/ in your browser and go to User Accounts > Add User Account.



Connect to your database server

- Fill out the information as shown below. Create a user name and password. The host name should be "localhost." You can "Check all" for privileges.
- Leave SSL option to "Require None"
- Press the "Go" button at the bottom right to complete the new user creation. You have just made yourself a superuser on localhost server!



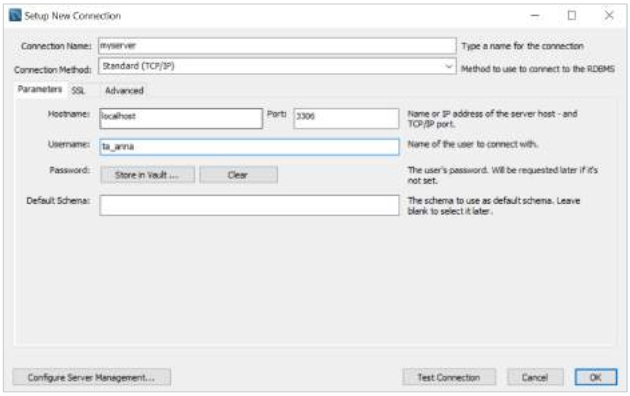


Connect to your database server

- Go back to Workbench and click the "+" icon next to "MySQL Connections" to connect to your database server with Workbench.

Connect to your database server

- Name the connection "myserver," enter "localhost" for host name, your username, and your MySQL port number you checked in MAMP (probably 3306).
- If you want Workbench to store your password, you can click the "Store in Vault" button, and enter the password you created in the previous step.
- Make sure the connection works by clicking "Test Connection" in the lower right of the window. After entering the password, it should say "Successfully made the MySQL Connection."
- Click Ok, to create the connection.

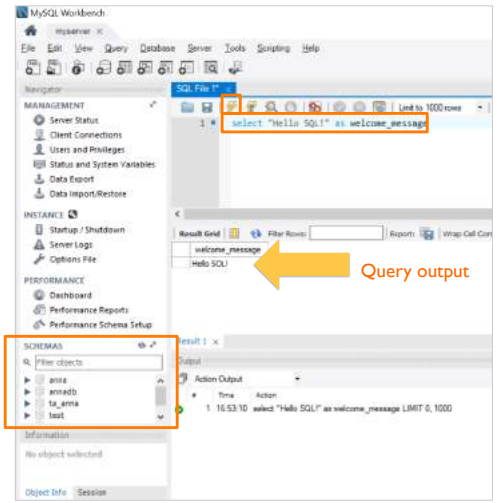


Connect to your database server

- Double click on your new connection to open it.

Connect to your database server

- You should see a list of "schemas" at the left, which should be the same as those displayed on the left-hand panel of phpMyAdmin.
- The middle panel of the screen should be a blank query window. (If not, click the +SQL ( ) icon under the File menu).
- Enter the example query shown here, and then press the lightning bolt icon to execute it.
- You've run your first SQL query!

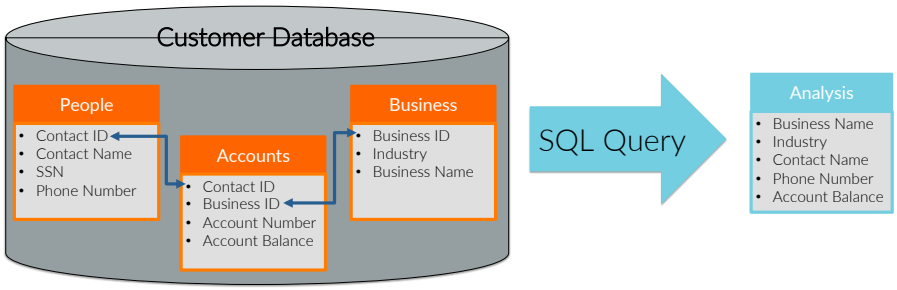


Outline

1. Overview of SQL & importing data
2. Connecting to your database in Python
3. Adjusting tables and foreign keys
4. Creating tables in a database
5. Performing basic statements in SQL
6. Performing UNIONS and JOINS in SQL
7. Advanced: using ORMs for larger databases

Intro to relational databases

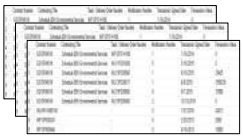
- What is a relational database?
  - Relational databases store data in the form of tables that can be related to one another based on common attributes in the columns and rows of those tables
  - SQL Queries can leverage these relationships to rearrange the data stored in database tables



SQL Server general components

- Servers
  - Server – database servers are programs that provides database services to other computer programs

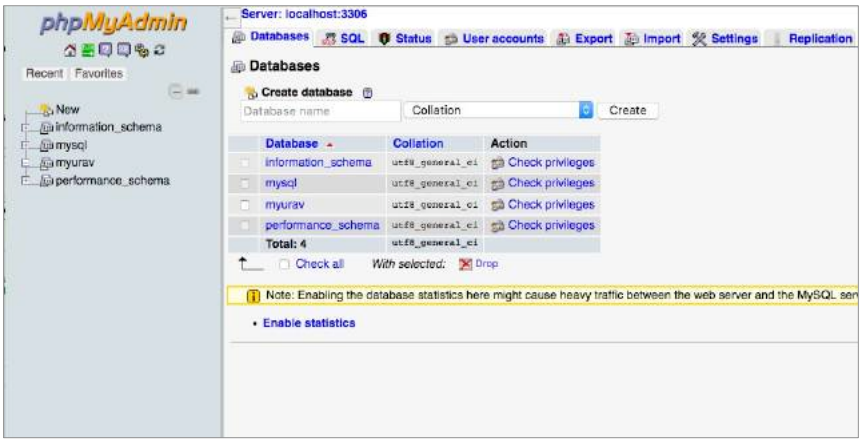
- Databases
  - Database – is a container of data/ information organized into tables (and other structures) so that they can be easily managed and accessed back in same fashion.



- Tables
  - Table – data stored in a tabular format with rows of named columns

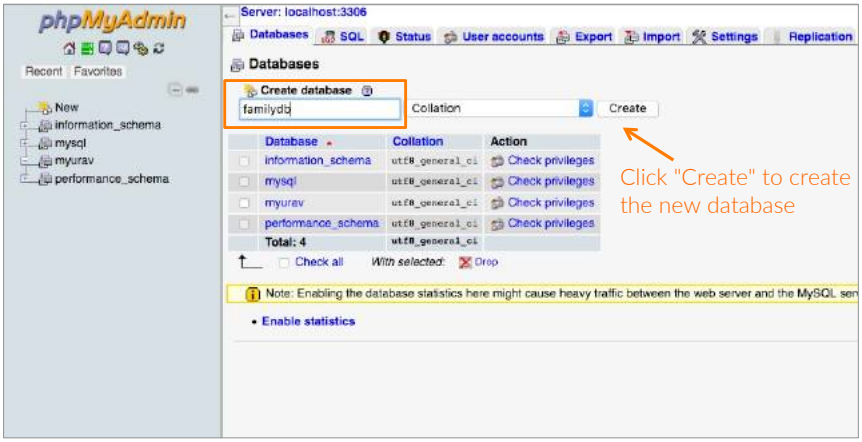
Creating a database

- First, go to http://localhost/phpMyAdmin/



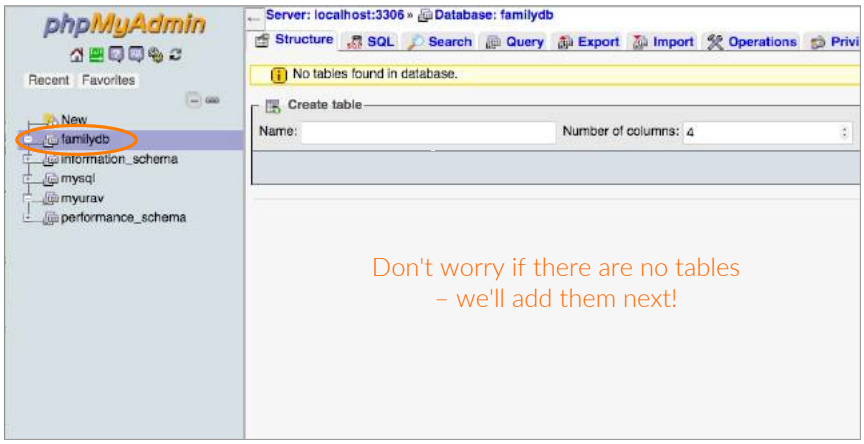
Creating a database

- Under "Create database", type in 'familydb', and click "Create"



Creating a database

- You should see the new database on the left-hand side



Populating a database

In order to practice SQL queries, you will need to run the ***familydb.py file*** *in the 'family' folder. Please refer to the appendix for more detailed information about building SQL tables in Python.*

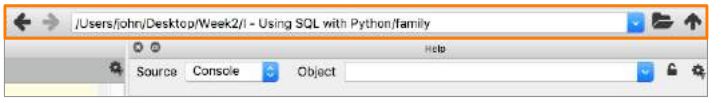
Connecting to a database

familydb.py

```
# First, import the MySQL connector library if you haven't already.
import mysql.connector

# Then, connect to an existing database:
dbconn = mysql.connector.connect(user='MYUSERNAME',
                                 password='',
                                 host='127.0.0.1',
                                 database='familydb')
```

1. Type in your username
2. Leave blank if you didn't use a password
3. Type in the local host
4. Type in the database name

Make sure your Spyder working directory is set to the `family` folder to pull data from it – then run the familydb.py file



Running SQL in Python

familydb.py

```
# The dbconn variable has a MySQLConnection type. This object holds
# the connection to the database. To run commands or SELECT statements,
# we need to create cursor object from this connection:
cur = dbconn.cursor()

# The cursor allows us to run SQL statements with the execute command:
cur.execute('select * from person')
cur.fetchall()

# To select the first result, type:
cur.fetchone()

# To select a number of results, type:
cur.fetchmany(n)
```

'n' stands for the number of results you want

If you execute a statement that generates a result, you must fetch the results before executing another statement. If you don't, you will see the error: ***InternalError: Unread result found.***

commit() for database operations

familydb.py

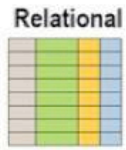
```
# In addition to retrieving data, we can also run database operations.
# These operations will change something about the database.
cur.execute('DROP TABLE family')

# However, the DROP command is "pending" until we run commit().
dbconn.commit()
```

This is a method of dbconn, not cur

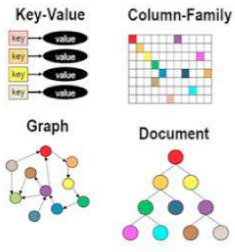
If you do not run commit(), the changes will be 'rolled back' once the session is ended.

Relational vs non-relational databases

- Relational database: consists of various tables that have explicit relationships to each other
  - Uses less disk space, but needs a schema (roadmap) for the tables
  - Uses SQL to query data, edit it, and rearrange it



- Non-relational databases (NoSQL): document-oriented databases with non-structured data that isn't necessarily categorized into fields
  - Takes up more storage than relational databases, but is becoming more popular given the decreasing cost of storage
  - Simpler and faster queries



Images from http://bigdata.iexpertify.com/wp-content/uploads/2013/09/NoSQL.jpg

MySQL syntax

- MySQL is just one of many different database management software products on the market. Others include Microsoft SQL Server, Oracle, PostgreSQL, DB2, Teradata, and SQLite.
- Most of what we show you here for MySQL will work with a different database management system, but possibly with a slight change to syntax.
- We will be using the MySQL Connector library for Python

Tables and data types

- Each table in a MySQL database is defined as follows:

SQL

```
CREATE TABLE family(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    last_name VARCHAR(50) NOT NULL
)
```

Defines the table name as 'family'

ID — Data type — Define column — Don't allow null values — Generates incrementing ID

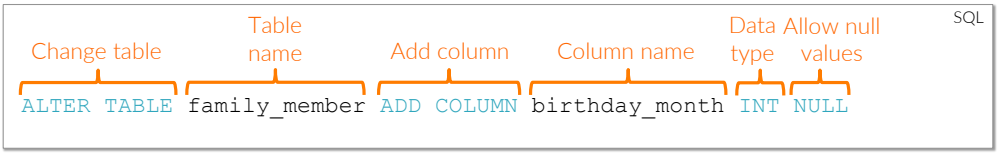
Defines the columns in the table

- The "AUTO_INCREMENT" notation tells MySQL to auto-generate an incrementing number starting from 1 to generate an 'id' column for a primary key
- The "full address" of this table is `familydb.family` – referencing the `family` table from the `familydb` database

Adding or dropping columns

- Adding columns to an existing table:

SQL



- Dropping columns from an existing table:

SQL

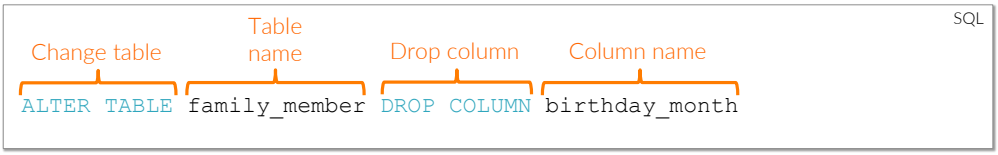


Table constraints in brief

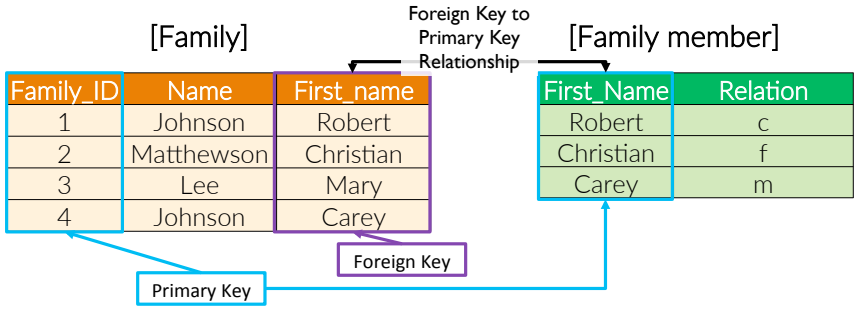
- Constraints are sets of rules imposed on the columns of a table or the table as a whole to limit the type of data going into that table.
- Constraints help insure the accuracy and integrity of a database and it's tables

Constraint Examples

- PRIMARY Key - a column (or columns) that serve as unique identifier for each row of data in a table consisting of 1 or more fields. Primary keys must not have null or duplicate values
- FOREIGN Key - a column (or columns) that corresponds to a primary key in another table acting as a cross reference between tables
- NOT NULL – ensures that a column does not contain NULL values
- DEFAULT – Substitutes a default value for a column when no value is provided
- UNIQUE - Ensures all values in a column are different
- INDEX – Stores indexed values from one or more columns to retrieve data quickly from a table

Foreign key relations

- Foreign keys ensure and enforce a relationship between tables
  - i.e. if we have a `family_member` table and add a foreign key constraint, we can't add a record to it that is not cross-referenced in the main `family` table
  - So every record in the `family_member` table is guaranteed to have a corresponding reference in `family`



Foreign key relations

- The syntax for creating a foreign key is shown below:

SQL

```
CREATE TABLE family_member(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    first_name varchar(50) NOT NULL,
    relation CHAR(1) NOT NULL,
    birth_date DATE NOT NULL,
    current_age INT NOT NULL,
    family_id INT NOT NULL,
);

ALTER TABLE family_member
 ADD CONSTRAINT fk_family
    FOREIGN KEY (family_id)
        REFERENCES family(id)
        ON DELETE CASCADE
;
```

Defining the columns of the table `family member`

1. Specifies which table to alter
2. Identify the name of the constraint
3. Identify the column for foreign key
4. Identify the table (column) to refer to

Tells SQL to delete entries from the child table if they're deleted from the parent table

Dropping foreign keys

- When you are creating and dropping tables, you need to keep in mind any existing foreign keys
- If you want to drop a parent table without deleting the children tables, you have to drop the foreign key first
- Below is the syntax you can use to drop a foreign key:

```SQL
ALTER TABLE family_member DROP FOREIGN KEY fk_family;
```


Outline

1. Overview of SQL & importing data
2. Connecting to your database in Python
3. Adjusting tables and foreign keys
4. Creating tables in a database
5. Performing basic statements in SQL
6. Performing UNIONS and JOINS in SQL
7. Advanced: using ORMs for larger databases

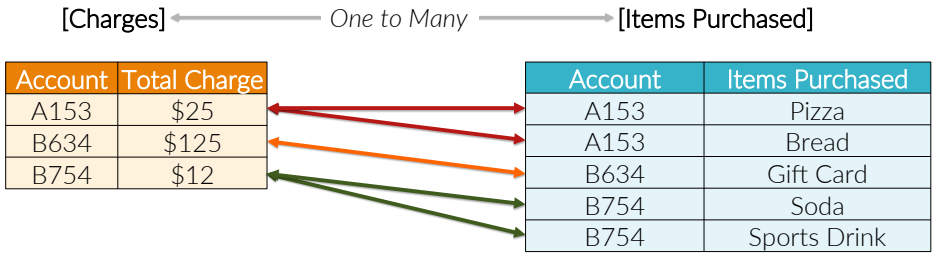
Relationship types

There are several types of relationships between table records. These relationships include:

- One-to-One - each record in one table will have no more than one matching record in a second table, and vice versa.
- One-to-Many - each record in one table can have many matching records in a second table; however, each record in the second table can only have one matching record in the first table.
- Many-to-Many – records in one table can have many matching records in a second table, and vice versa.

One-to-Many relationships

One-to-Many relationships exist when each records in one table may relate to numerous records in another table.



Account	Total Charge
A153	$25
B634	$125
B754	$12

Account	Items Purchased
A153	Pizza
A153	Bread
B634	Gift Card
B754	Soda
B754	Sports Drink

Many-to-Many relationships

Many-to-Many relationships exist when records in one table can have many matching records in a second table, and vice versa.

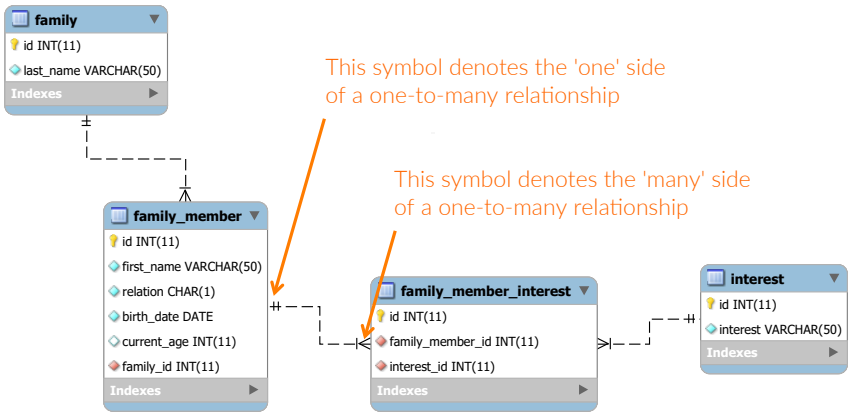
[Authors] ⟵ *Many to Many* ⟶ [Publications]

Author	Location
John	LA
Merav	MD
Dmitri	VA
Katya	VA

Article	Views
War Time Human Right Violations	50
Trends in Modern Contract Law	100
Data Driven Enterprise Decisions	75
Perils of Investment Banking	25
Advanced Text Mining Techniques	30

The entity-relation diagram

- Here is a diagram of the database we built



Inserting data into a table

- If you want to add rows to a table by hand, you can write:

Insert a value into table (column). The value is 'clarke'.

```SQL
INSERT INTO family(last_name) VALUES ('clarke')
```

- Note: we don't need to specify the id column – for every column not specified, MySQL will place either a NULL value or default value

- Since we defined the auto incrementing id value, the new row generated will have the next value in the sequence

Creating tables from a query

- It is often useful to create a table and populate it with the result of a query:

```SQL
CREATE TABLE family_member_first_name
AS
SELECT id, first_name
FROM family_member
```

1. Table to create
2. Values to select
3. Table to grab values from

- This creates columns of the new table 'family_member_first_name', based on the columns selected from the query, 'id' and 'first_name'

Note: this only imports the values, it does not have the same foreign keys, indexes, or column default values

Temporary tables

- Temporary tables are useful to store intermediate results of a complex calculation

```sql
CREATE TEMPORARY TABLE result1
AS
SELECT ...
```

SQL

1. Table to create
2. Values to select

- Typically, these tables are created as a result of a query
- The lifetime of a temporary table is the same as your current connection to the database – once it's disconnected, the tables are automatically dropped

Truncating and dropping tables

- To remove all the data in a table, use the `TRUNCATE` command

```sql
TRUNCATE family_member_interest
```

SQL

Truncate — Table name

- Drop columns from an existing table with the `DROP` command

```sql
DROP TABLE family_member_interest
```

SQL

Change table — Table name

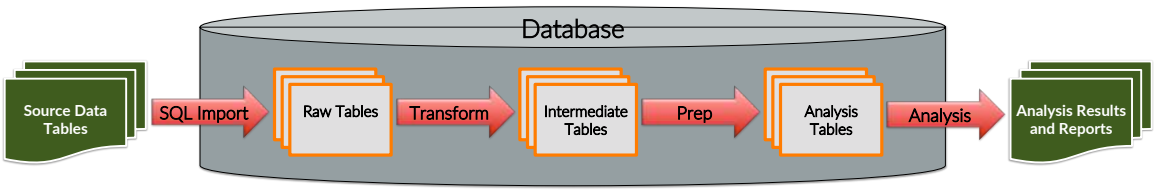
- If you run the above command a second time, you'll receive an error from MySQL since the table no longer exists – if you want to ensure that the table doesn't exist, you can add the `IF EXISTS` syntax

```sql
DROP TABLE IF EXISTS family_member_interest
```

SQL

3 stages of data tables

- Often tables imported from another environment are not ready for analysis
- When preparing for an analysis think of three types of tables:
  - Raw – immediately imported
  - Intermediate – some data wrangling has been performed
    - Ex: dates might be imported as text and require transformation into a date format for analysis
  - Analysis – table has been aggregated and prepared for analysis



*Note: it is a best practice to name tables to distinguish these table types (Ex. tbl_Raw_Procurement_Data _2016)

Outline

1. Overview of SQL & importing data
2. Connecting to your database in Python
3. Adjusting tables and foreign keys
4. Creating tables in a database
5. Performing basic statements in SQL
6. Performing UNIONS and JOINS in SQL
7. Advanced: using ORMs for larger databases

SQL data types

There are many different data types in SQL Server; however, there are 3 main data type categories:

- Numeric: contains numbers and can be used in mathematical operations
- Character: contains strings of text and can be searched for words and phrases or concatenated
- Date: contains dates and/or times that are stored as number allowing date type fields to also be used in mathematical operations

Data type examples

Numeric	Character	Date
1	A123F	1/01/2000
-2,000	Coffee is a great way to start off your day	2005-07-01 00:00:00:000
$250.35	Automobile	June 16 2013
0.0023464	Desk	Monday, January 31 2002

Examples of data types

Numeric Data Types
- Int
- Money
- Decimal

Character Data Types
- Char
- Varchar
- Nvarchar

Date Data Types
- Datetime
- Date
- Time

https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-types-transact-sql

NULLs

- NULL values are non-existing records in a field. They are different from "blank" or "zero-length" string values (i.e. “”)
  - NULL values are excluded from aggregate functions:
    - Example: when SQL counts the number of records in the `ID_Number` column (“`COUNT(ID_Number)`”) it returns a count of 2
  - NULL values do not link to one another when they are in a field being used as the relationship for combining tables
- To locate `NULL` values, use “`IS NULL`” (or “`IS NOT NULL`”) in a `WHERE` clause
  - Example: “`WHERE ID_Number IS NULL`” would return only row 3 in the table below

BLANK Values

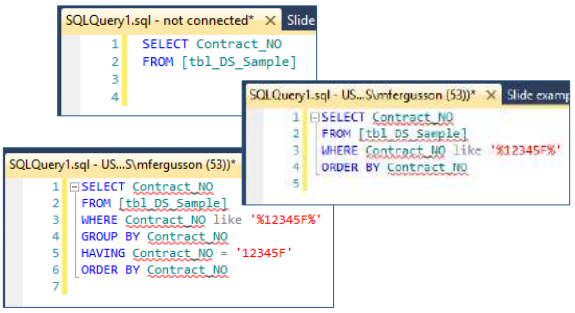
NULL Values

	ID_Number	Category	Purchase_Date
1		BMW	2002-01-01 00:00:00.000
2	1112	Mercedes	NULL
3	NULL	NULL	1970-12-01 00:00:00.000

Query examples and explanation

- SQL Queries follow a standard order of statements that must be followed in each query for SQL server to understand a query
  - Not all statements are required for every query, but the same order of commands must be maintained

1. SELECT
2. INTO
3. FROM
4. WHERE
5. GROUP BY
6. HAVING
7. ORDER BY



The SELECT statement

- To retrieve data from a single table, the query syntax is:

```
SELECT column 1, column 2,...
FROM table
WHERE logical conditions
ORDER BY column1, column2 desc
LIMIT N
OFFSET M
```

SQL

1. SELECT – defines the fields that will be included in the new table requested by a query from other tables and from functions
2. FROM – defines the existing tables that a query will draw data from
3. WHERE – filters the query results based on criteria from the original tables
4. ORDER BY – sorts the query results in order by the indicated fields
5. LIMIT – limits the number of records based on the value
6. OFFSET – excludes the first M number of records

The SELECT statement

- This query selects the five youngest members from the 'family_member' table

```sql
SELECT * <-- You can use * to select all columns
FROM family_member
ORDER BY current_age
LIMIT 5
```

- The database does not necessarily store the rows of a table in any particular order - different queries performed by you or other users may reorder the rows
- To make sure the rows are returned in a particular order, specify an ORDER BY condition.

WHERE clause logical conditions

- Below are some of the most common logical statements used in SQL `WHERE` clauses

```sql
WHERE...
  column1 < column1
  column1 = column2
  column1 <> column2 <-- Checks if they are not equal
  column1 <= column2
  column1 between column1 and column2
  column1 in (10, 14, 28, 30) <-- Checks if value of column1 is in a set of values
  column1 not in (10, 14, 28, 30)
```

WHERE clause logical conditions

- Similar to Python, logical statements can be combined with `AND` or `OR`:

```sql
WHERE column1 > 7 AND column2 = 'a'
```

SQL

- More complicated conditions can be applied with parentheses:

```sql
WHERE (column1 < 12 AND column2 = 'a')
OR (column1 >= 12 AND column2 = 'b')
```

SQL

Be very careful when using OR with a compound logical statement such as this one. Remember, statements inside parentheses are evaluated first.

Changing values with UPDATE

- We can modify specific values with an `UPDATE...SET` command

```sql
UPDATE family_member
SET relation = 'mother'
WHERE  relation = 'm'
```

SQL

1. Update the specified table
2. Instructs the column to modify and the new value to give it
3. Specifies which values to modify

- We can have SET use a calculation based on another column – here, we will create a new birthday month column and populate it based on birth_date

```sql
ALTER TABLE family_member ADD COLUMN birthday_month INT NULL
UPDATE family_member
SET
```

SQL

Deleting rows from a table

- You can use `DELETE` to identify specific rows that you want to remove

```sql
DELETE FROM interest
WHERE interest = 'reading'
```

← removes reading from the interest table

The CASE statement

- You can use a CASE statement to derive a new column based on the values of other columns

```sql
SELECT first_name,
  current_age,
  CASE WHEN current_age < 18 THEN 'child' else 'adult' END
  AS age_type
FROM family_member
```

← New column name defined by AS

- Providing a new name with AS is called *aliasing* a column – any column can be aliased this way, even those that are not derived

The CASE statement

- CASE statements can also provide a custom sort order

```SQL
SELECT family_id, first_name, relation
FROM family_member
ORDER BY family_id,
  CASE relation
    when 'm' then 1
    when 'f' then 2
    when 'c' then 3
  END
```

Here, we're sorting by m, f, c

- If we had only ordered by 'family_id' and 'relation', the relations would be sorted in alphabetical order (c, f, m)

Aggregating data

- Oftentimes, a column will have fewer distinct values than the total number of rows. It is then possible to summarize by this column.

```SQL
SELECT *
FROM family_member
ORDER BY family_id, birth_date
```

Aggregating data

id	first_name	relation	birth_date	current_age	family_id
2	francesca	m	1964-11-18	52	1
1	mark	f	1965-03-16	52	1
3	thomas	c	1990-12-09	26	1
4	mary	c	1992-08-21	25	1
5	helen	c	1994-10-14	23	1
6	thomas	f	1982-12-07	34	2
7	catherine	m	1983-08-13	34	2
8	anthony	c	2010-04-15	7	2
9	patricia	c	2012-05-17	5	2
10	joseph	c	2014-03-14	3	2
11	jim	f	1988-09-01	29	3
12	theresa	m	1989-07-06	28	3
13	adam	c	2016-01-04	1	3
14	charles	f	1969-05-21	48	4
15	cynthia	m	1970-11-18	46	4
16	angela	c	1995-06-01	22	4
17	michael	c	1997-05-29	20	4
19	roseanne	m	1968-04-17	49	5
18	stephen	f	1969-12-15	47	5

We can notice that there are fewer distinct values of the 'family_id' column

Aggregating data

- We can treat each distinct value of the family_id as a "group", and compute calculations on each group. For example, to get the total number of members in each group, we would write:

SQL

```
SELECT family_id, count(*) as num_members
FROM family_member
GROUP BY family_id
```

← Tells MySQL how to define the groups

family_id	num_members
1	5
2	5
3	3
4	4
5	5

Additional aggregation functions

- Other *aggregation functions* you can use are sum(), avg(), max(), and min(). For example:

SQL
```
SELECT family_id, count(*) as num_members,
  max(current_age) as oldest_age,
  avg(current_age) as avg_age,
  min(current_age) as youngest_age
FROM family_member
GROUP BY family_id
```

- You can go here for additional functions: https://dev.mysql.com/doc/refman/5.5/en/group-by-functions.html

Group by multiple variables

- You can group by multiple variables as well. This defines groups and subgroups. If we get a distinct family_id and a distinct relation, for example, then there is one group for each unique combination of family_id and relation.

SQL
```
SELECT family_id, relation, count(*)
FROM family_member
GROUP BY family_id, relation
```

- Or we can ask "How many people have birthdays in each month?"

SQL
```
SELECT EXTRACT(month from birth_date) as bday_month,
count(*) as num_people
FROM family_member
GROUP BY 1
ORDER BY 1
```

Extract the month from the birth date

Group by multiple variables

Note that only the months that appear in the data will show up – there are no February birthdays, so there is no birthday month 2.

bday_month	num_people
1	1
3	2
4	2
5	3
6	2
7	1
8	2
9	1
10	1
11	3
12	4

What does the * mean in count(*)?

- Count, like the other aggregation functions, can be given a column as an argument, but it won't count the nulls
- You can use `count(*)` to count all of the *rows* in the group, as opposed to values in a particular column. This may not seem important now, but it will useful when we learn about outer joins.

Aggregation with WHERE

- We might ask "how many members of each family are over age 30?"

SQL

```
SELECT family_id, count(*)
FROM family_member
WHERE current_age >= 30
GROUP BY family_id
```

- Filtering by HAVING is helpful if we want to filter after we have computed the aggregate?

SQL

```
SELECT EXTRACT(month from birth_date) as bday_month,
FROM family_member
GROUP BY 1
HAVING count(*) >= 2
```

Here, we're extracting the months that have at least two birthdays

Combining aggregations

- What if we want to answer these three questions:
  - How many members of each family are over age 30?
  - How many children in each family are older than 4?
  - How many members of each family have a birthday in December?

SQL

```
SELECT family_id,
 sum( case when current_age >= 30 then 1 else 0 end ) as gt30,
 sum( case when relation = 'c' and current_age > 4 then 1 else 0 end) as cgt4,
 sum( case when EXTRACT(month from birth_date) = 12 then 1 else 0 end) as bday12
FROM family_member
GROUP BY family_id
```

family_id	gt30	cgt4	bday12
1	2	3	1
2	2	2	1
3	0	0	0
4	2	2	0
5	2	3	2

Outline

1. Overview of SQL & importing data
2. Connecting to your database in Python
3. Adjusting tables and foreign keys
4. Creating tables in a database
5. Performing basic statements in SQL
6. Performing UNIONS and JOINS in SQL
7. Advanced: using ORMs for larger databases

Unions

- UNION combines two or more select statements 'by rows' – the statements must contain the same number of columns and data types

SQL

```
SELECT first_name as "First Name",
  birth_date as "Date of Birth",
  relation as "Relation to Family"
FROM family_member
WHERE relation = 'm'

UNION

SELECT first_name, birth_date, relation
FROM family_member
WHERE relation = 'f'
```

Note: This example is a bit contrived because we could have just as easily gotten the same result using a single query with a WHERE clause. The typical use case for UNION is when the two record sets are in different tables.

Unions

- It is extremely important that the column names match up, otherwise the output may not be what you expect

SQL

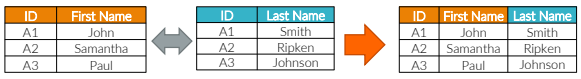
```
SELECT first_name, birth_date, relation
FROM family_member
WHERE relation = 'm'

UNION

SELECT relation, birth_date, first_name
FROM family_member
WHERE relation = 'f'
```

This is a different column order than above

SQL table combinations

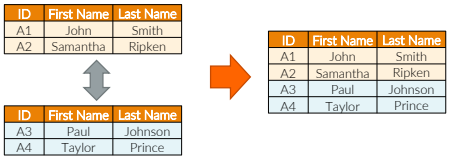
SQL tables can be combined using `JOIN` or `UNION` statements to merge columns or records respectively

JOIN



- A `JOIN` brings columns from 2 different tables into a combined table
- The combined table can have more or fewer records than its parent tables depending on both the relationship between records on the parent tables and the type of join performed

UNION



- A `UNION` appends records from 2 tables into a combined table
- The combined table will have more records than its parent tables assuming no filtering is applied to either table

Using table aliases

A table alias allows a table to be temporarily renamed within the scope of an individual query. This makes queries easier to read and limits the amount of code required for a query to execute.

Sample

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	ID	Last Name
A1	John	A1	Smith
A3	Paul	A3	Johnson

Code

```
SELECT a.id as family_member_id,
  b.id as family_id, first_name, last_name, birth_date
FROM family_member as a
JOIN family as b
    ON a.family_id = b.id
```

The aliases of "a" and "b" are used to replace tables `family_member` and `family` respectively for indicating the source of each field in the query

Aliases can be assigned with or without "AS" following a table name

More complex join conditions

- Suppose we wanted to show pairs of individuals where for each person, we show everyone who is younger than that person
- The following example uses a "self join" - joining a table to itself – to answer this question.

SQL

```
SELECT a.id as id_younger, a.first_name as first_name_younger,
  a.current_age as current_age_younger,
  b.id as id_older, b.first_name as first_name_older,
  b.current_age as current_age_older
FROM family_member as a
JOIN family_member as b
ON a.birth_date > b.birth_date
ORDER BY a.birth_date desc, a.first_name, b.birth_date desc
```

Outer joins

- It might not always be the case that every value in the first ("left") table matches a value in the second ("right") table:

```SQL
SELECT family_member_id, first_name, interest_id
FROM family_member as a
JOIN family_member_interest as b
 on a.id = b.family_member_id
order by family_member_id
```

- Here, the results show us that we're missing person numbers 1, 5, 19, and 22 are missing from this table
- An *outer join* (as opposed to *inner join*) will retain every value in the left table
- If there is no match on the right, we will still see the value from the left, and all of the right-hand side column will have a value of NULL.

Left joins

- The only difference to perform an outer join is to replace JOIN with LEFT OUTER JOIN, or simply LEFT JOIN:

```SQL
SELECT a.id, first_name, interest_id
FROM family_member as a
LEFT JOIN family_member_interest as b
 on a.id = b.family_member_id
 order by family_member_id
```

Note with an outer join, it matters which table we use in the order by!

- Note, there is also a right outer join, but you just need to switch the order of the two tables and using a left join. For this reason it is hardly used.

Left joins

- If we wanted to get a count of each person's interests, we could run the following code:

```SQL
SELECT a.id, count(interest_id) as num_interests
FROM family_member as a
LEFT JOIN family_member_interest as b
 on a.id = b.family_member_id
GROUP BY a.id
ORDER BY a.id
```

- The right-hand side column `interest_id` will be null in the case of no match, and count(x) only counts non-null values of the column x.

Left joins: more advanced

- Francesca and Mark are both 52 years old, which is the oldest age in the `family_member` table
- Since Francesca is slightly older, she does not appear on the left hand side of our results because there was no match on the right hand side for her, i.e., there was no one found who is older than her
- So we can change the query to do a left join instead:

```SQL
SELECT a.id as id_younger, a.first_name as first_name_younger,
  a.current_age as current_age_younger,
  b.id as id_older, b.first_name as first_name_older,
  b.current_age as current_age_older
FROM family_member as a
LEFT JOIN family_member as b
 ON  a.birth_date > b.birth_date
ORDER BY a.birth_date desc, a.first_name, b.birth_date desc
```

Cross joins

- A cross join is simply an inner join, where the `ON` condition is always true

```SQL
SELECT a.last_name, b.last_name
FROM family as a
JOIN family as b
 ON 1 = 1
```

Always true, so every row in table a will be matched with every row in table b

If you take the cross join of two tables A and B, and table A has 500 rows and table B has 600 rows, how many rows will you get back?

Subqueries

- You can use a query, surrounded by parentheses, as a pseudo-table, such as in the following example:

```SQL
SELECT a.id, last_name, min_age, max_age
FROM family as a
JOIN
 (SELECT family_id, min(current_age) as min_age, max(current_age) as max_age
    FROM family_member
    GROUP BY family_id) as b
ON a.id = b.family_id
```

"Table b" is called a subquery

- Subqueries are extremely useful for combining information on the fly, but can become computationally expensive – it may be better to create a temporary table

*Note: you **must** provide an alias for the subquery (for example 'as b' above, otherwise you will get an error: "Every derived table must have its own alias."*

Multiple joins

- It is extremely important that the column names match up, otherwise the output may not be what you expect

SQL

```
SELECT first_name, birth_date, relation
FROM family_member
WHERE relation = 'm'

UNION

SELECT relation, birth_date, first_name
FROM family_member
WHERE relation = 'f'
```

This is a different column order than above

Indexing for better query performance

- It is extremely important that the column names match up, otherwise the output may not be what you expect

SQL

```
SELECT first_name, birth_date, relation
FROM family_member
WHERE relation = 'm'

UNION

SELECT relation, birth_date, first_name
FROM family_member
WHERE relation = 'f'
```

This is a different column order than above

Indexing for better query performance

- It is extremely important that the column names match up, otherwise the output may not be what you expect

```sql
SELECT first_name, birth_date, relation
FROM family_member
WHERE relation = 'm'

UNION

SELECT relation, birth_date, first_name
FROM family_member
WHERE relation = 'f'
```

SQL

This is a different column order than above

Getting metadata from information_schema

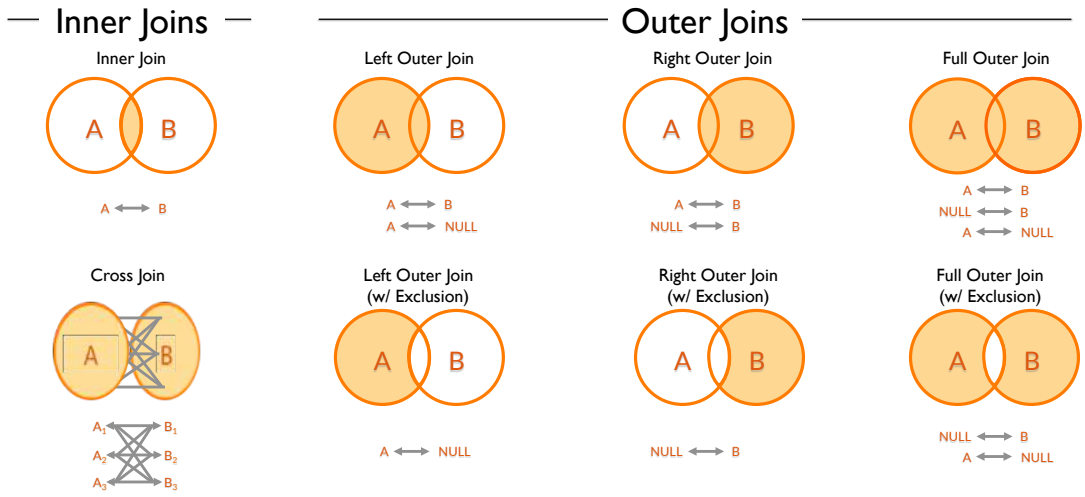
- It is extremely important that the column names match up, otherwise the output may not be what you expect

```sql
SELECT first_name, birth_date, relation
FROM family_member
WHERE relation = 'm'

UNION

SELECT relation, birth_date, first_name
FROM family_member
WHERE relation = 'f'
```

SQL

This is a different column order than above

Joining logic comparison



Outline

1. Overview of SQL & importing data
2. Connecting to your database in Python
3. Adjusting tables and foreign keys
4. Creating tables in a database
5. Performing basic statements in SQL
6. Performing UNIONS and JOINS in SQL
7. Advanced: using ORMs for larger databases

Quality control tips for joins

- Pay attention to the row counts!
  - Is the number of rows what you expect?
    - A common problem is assuming that a field or combination of fields represents a unique value. When that is not the case you can see an increase in record count.
    - Another common issue is incorrectly setting criteria of a join or WHERE clause and excluding more records than were originally intended.
  - Use simple queries on each original table to compare the number of records with given criteria to the number in the joined table.
  - It is also helpful to use LEFT and/or RIGHT joins to determine the overlap between tables and whether:
    - The fields you are using to link the original tables are appropriate.
    - The tables themselves are appropriate to combine for an analysis (2 tables can appear to be similar data from field names but actually contain little to no overlap).

Logical operators: comparisons

Logical operators test whether or not a condition is true. They are generally used in `CASE` Statements, `JOIN`s, `WHERE` clauses, `HAVING` clauses.

Comparison Operators:

Operator	Operator name	Description
=	Equal	True when the 1st value equals the 2nd value being tested
<>	Not equal	True when the 1st value does not equals the 2nd value being tested
!=	Not equal	
<	Less than	True when the 1st value is less than the 2nd value being tested
<=	Less than or equal to	True when the 1st value is less than or equal to the 2nd value being tested
!>	Not greater than	
>	Greater than	True when the 1st value is greater than the 2nd value being tested
>=	Greater than or equal to	True when the 1st value is greater than or equal to the 2nd value being tested
!<	Not Less than	

Logical operators: comparisons

Logical operators test whether or not a condition is true. They are generally used in `CASE` Statements, `JOIN`s, `WHERE` clauses, `HAVING` clauses.

Comparison Operators:

Operator	Description
IN	True when values being tested match values in a list of values specified either as a query resulting in 1 column or a list of values separated by commas
BETWEEN	True when values being tested are within an inclusive range (lower and upper values are searched as well as the values in between)
LIKE	True when values being tested match a patterns in character fields specified after the LIKE operator
NOT	used to reverse the logic of the IN, BETWEEN, or LIKE operators to negate conditions (i.e. a query containing " WHERE NOT LIKE '%data%' " would return all records that do not contain "data" anywhere in the string being tested)

Logical operators: LIKE

The LIKE clause matches patterns of text to a character field. It is important to understand how to leverage special characters ("%" , "_", "[", "]", and "^") to accurately match patterns.

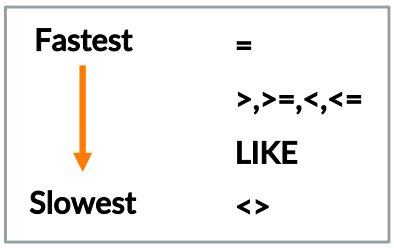
- Wildcard - characters that substitute for any other character in a string
  - "%" - used to represent zero or more characters and is typically used before and/or after the part of text being searched for to look for that text anywhere in the character string
  - "_" - used to represent 1 character
- Specified patterns - Brackets "[]" can be used to specify lists or ranges of characters or numbers that should be represented by a character in a pattern
  - Using "^" after the opening bracket changes exclude the characters following it (ex. "[^m]" matches any letter other than "m")
- Escape - The special characters listed above ("%" , "_", "[", and "]") need to be treated differently than others. They either need to be included in brackets or placed after an escape character

Multiple logical operators

Multiple logical operators can be strung together by relating them with `AND` or `OR` statements.

- `AND` - used to connect two or more conditions and only returns those rows meeting all conditions
- `OR` - used to connect two or more conditions and returns any rows that meet any of these conditions
- `Order of logical operations:` "( )" → "AND" → "OR"

Performance: argument order



- Try to use a leading character with `LIKE` (ex. `LIKE 'm%'` instead of `LIKE '%m'`)
- Use LIKE instead of SUBSTRING with =

OR
AND

- Be careful with `OR`
- If multiple `AND`s, put least likely condition first
- If equally likely, put least complex condition first

Performance: query structure

- Restrict result sets by using WHERE or only selecting the columns needed
- Use WHERE with HAVING when appropriate
- ORDER BY is inefficient; sort results in a separate step

Performance: space efficiency

- Backup your database, but don't store excessive copies of the backup
- Choose between temp tables and views
- Use appropriate field types (avoid using NVARCHAR)

Questions?

APPENDIX

Creating tables in database

familydb.py

```
# Next, we will populate our database with tables.
tables=[
{
'name': 'family',
'query': """
create table family(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    last_name VARCHAR(50) NOT NULL
)
"""
},
```

ID — Data type — Define column — Don't allow null values — Generates incrementing ID

1. Name of the table
2. Sets up the query
3. Create the table 'family'
4. These lines name and define attributes of the fields that will populate the empty tables being created

SQL

```
CREATE TABLE family(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    last_name VARCHAR(50) NOT NULL
)
```

Example of what the code translates to in SQL

Creating tables in database

familydb.py

```
# ...Continued from the previous slide...
{
'name': 'interest',
'query': """
create table interest(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    interest VARCHAR(50) NOT NULL
)
"""
},
```

ID — Data type — Define column — Don't allow null values — Generates incrementing ID

1. Name of the table
2. Sets up the query
3. Create the table 'interest'
4. These lines name and define attributes of the fields that will populate the empty tables being created

SQL

```
CREATE TABLE interest(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    interest VARCHAR(50) NOT NULL
)
```

Table constraints in brief

- Constraints are sets of rules imposed on the columns of a table or the table as a whole to limit the type of data going into that table.
- Constraints help insure the accuracy and integrity of a database and it's tables

Constraint Examples

- PRIMARY Key - a column (or columns) that serve as unique identifier for each row of data in a table consisting of 1 or more fields. Primary keys must not have null or duplicate values
- FOREIGN Key - a column (or columns) that corresponds to a primary key in another table acting as a cross reference between tables
- NOT NULL – ensures that a column does not contain NULL values
- DEFAULT – Substitutes a default value for a column when no value is provided
- UNIQUE - Ensures all values in a column are different
- INDEX – Stores indexed values from one or more columns to retrieve data quickly from a table

Creating tables in database

familydb.py

```
# ...Continued from the previous slide...
{
 'name': 'family_member',
 'query': """
 create table family_member(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    first_name varchar(50) NOT NULL,
    relation CHAR(1) NOT NULL,
    birth_date DATE NOT NULL,
    current_age INT NOT NULL,
    family_id INT NOT NULL,
    )
"""
 },
```

1. Name of the table
2. Sets up the query
3. Create the table 'family member'
4. Here we define all the columns in the table

SQL

```
CREATE TABLE family_member(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    first_name varchar(50) NOT NULL,
    relation CHAR(1) NOT NULL,
    birth_date DATE NOT NULL,
    current_age INT NOT NULL,
    family_id INT NOT NULL,
);
```

Creating tables in database

familydb.py

```
# ...Continued from the previous slide...
{
 'name': 'family_member_interest',
 'query': """
 create table family_member_interest(
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    family_member_id INT NOT NULL,
    interest_id INT NOT NULL
    )
"""
 }
]
```

1. Name of the table
2. Sets up the query
3. Creates the table 'family_member_interest'
4. Defines all the columns in the table

Outline

1. Overview of SQL & Importing data
2. Connecting to your database
3. Creating tables in a database
4. Adding foreign keys and adjusting data
5. Performing basic statements in SQL
6. Performing UNIONS and JOINS in SQL
7. Advanced: using ORMs for larger databases

Importance of table relationships

- Combining data accurately from tables in a database or from disparate sources requires a thorough understanding of how fields in those tables are related
- Failure to understand these relationships can result in duplicate or missing records that can materially impact the results of your analysis

What are the total purchases from each state?

[Customers]

Cust_ID	Name	State
1	John	VA
2	Matt	VA
3	Lee	CA
4	John	MN

[Purchases]

Cust_ID	Name	Purch_ID	amt
1	John	1	$10.50
2	Matt	2	$5.00
3	Lee	3	$2.00
4	John	4	$4.25
1	John	5	$9.25

```
SELECT A.*
SELECT A.State, sum(B.amt) as State_Purchases
      FROM #Customers A
      JOIN #Purchases B
            ON A.Cust_ID = B.Cust_ID
      GROUP BY A.State
```

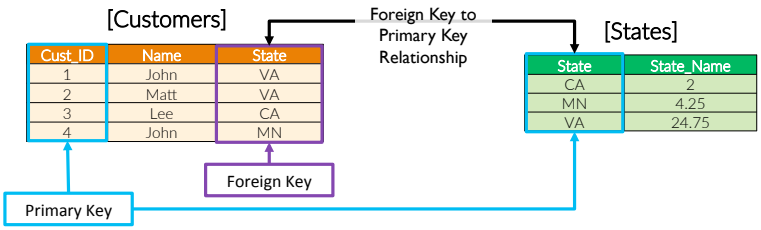
State	State_Purchases
CA	2
MN	4.25
VA	24.75

```
SELECT A.*
SELECT A.State, sum(B.amt) as State_Purchases
      FROM #Customers A
      JOIN #Purchases B
            ON A.Name = B.Name
      GROUP BY A.State
```

State	State_Purchases
CA	2
MN	24.00
VA	29.00

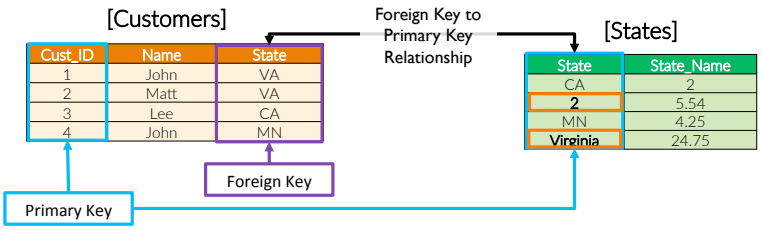
Table keys

- In order to maintain data quality and integrity between related tables, it is important to define those relationship through conceptual (or sometimes physical) constraints on table fields called keys.
- There are 2 main types of keys to note for this course:
  - Primary Key – a column (or columns) that serve as unique identifier for each row of data in a table consisting of 1 or more fields
    - Primary keys must not have null or duplicate values
  - Foreign Key – a column (or columns) that corresponds to a primary key in another table acting as a cross reference between tables



Inconsistent keys

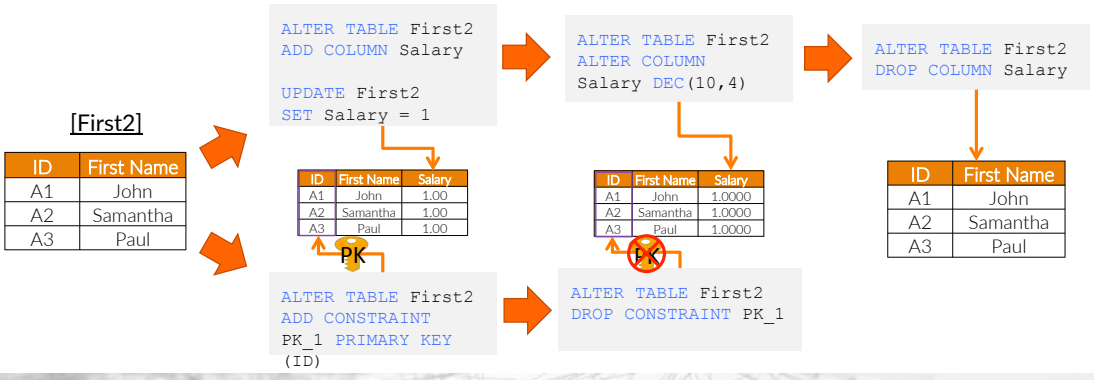
- It's important to remember that these key's should be well defined and structured as physical constraints of a database, but that is not always the case
- Often as an analyst you will encounter related tables where these key relationships are broken or not maintained properly
- Always check the data in related fields to make sure the relationship between 2 tables holds and that you are using the correct relationship to combine them



ALTER TABLE statements

The SQL ALTER TABLE statement can also be used to change data types as well as other modifications to the structure of a table such :as:

1. Adding, dropping, or modifying table columns
2. Adding, dropping constraints



Joins: code structure

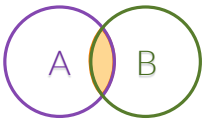
- The illustration below demonstrates how the `JOIN` code relates these tables. In more complex joins, tables are referred to as `LEFT` and `RIGHT` tables based on their order in the SQL statement, which will impact the records returned in result sets.

Code

```
SELECT A.*, B.[Last Name]
FROM [First] A
JOIN [Last] B
    ON A.ID = B.ID
```

The **LEFT** table follows **FROM**

The **RIGHT** table follows **JOIN**



Tables & Results

ID	First Name
A1	John
A2	Samantha
A3	Paul

[First] (LEFT TABLE)

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

[Last] (LEFT TABLE)

Result

ID	First Name	Last Name
A1	John	Smith
A3	Paul	Johnson

Joins: INNER JOIN

An `INNER JOIN` between 2 tables returns the intersection between those 2 tables

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	Last Name
A1	John	Smith
A3	Paul	Johnson

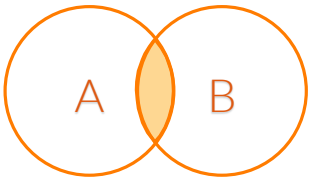
Code

```
SELECT A.*, B.[Last Name]
FROM [First] A
INNER JOIN [Last] B
    ON A.ID = B.ID
```

or

```
SELECT A.*, B.[Last Name]
FROM [First] A
JOIN [Last] B
    ON A.ID = B.ID
```

Logic



Connection

A ⟷ B

Joins: LEFT OUTER JOIN

A `LEFT OUTER JOIN` between 2 tables returns all records from the table in the initial table and the intersection between those 2 tables. Where there is no intersection NULL values are populated in columns selected from the joined table.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	Last Name
A1	John	Smith
A2	Samantha	*NULL*
A3	Paul	Johnson

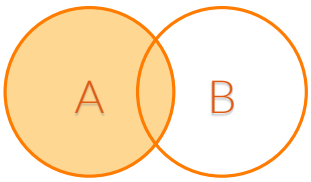
Code

```
SELECT A.*, B.[Last Name]
FROM [First] A
LEFT OUTER JOIN [Last] B
    ON A.ID = B.ID
```

or

```
SELECT A.*, B.[Last Name]
FROM [First] A
LEFT JOIN [Last] B
    ON A.ID = B.ID
```

Logic



Connection

A ⟷ B
A ⟷ NULL

Joins: LEFT OUTER JOIN (exclude)

A `LEFT OUTER JOIN` with exclusion between 2 tables returns only records from the original table with no intersection to the initial table.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	Last Name
A2	Samantha	*NULL*

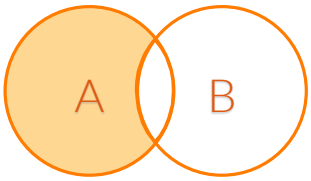
Code

```
SELECT A.*, B.[Last Name]
FROM [First] A
LEFT OUTER JOIN [Last] B
    ON A.ID = B.ID
WHERE B.ID IS NULL
```

or

```
SELECT A.*, B.[Last Name]
FROM [First] A
LEFT JOIN [Last] B
    ON A.ID = B.ID
WHERE B.ID IS NULL
```

Logic



Connection

A ⟷ NULL

Joins: RIGHT OUTER JOIN

A `RIGHT OUTER JOIN` between 2 tables returns all records from the joined table and the intersection between those 2 tables. Where there is no intersection NULL values are populated in columns selected from the table in the initial table.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	Last Name
A1	John	Smith
A3	Paul	Johnson
A4	*NULL*	Adler

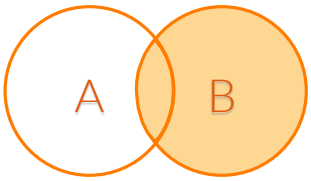
Code

```
SELECT B.ID, A.[First Name]
, B.[Last Name]
FROM [First] A
RIGHT OUTER JOIN [Last] B
    ON A.ID = B.ID
```

or

```
SELECT B.ID, A.[First Name]
, B.[Last Name]]
FROM [First] A
RIGHT JOIN [Last] B
    ON A.ID = B.ID
```

Logic



Connection

A ⟷ B

NULL ⟷ B

Joins: RIGHT OUTER JOIN (exclude)

A `RIGHT OUTER JOIN` with exclusion between 2 tables returns only records from the joined table with no intersection to the initial table.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	Last Name
A4	*NULL*	Adler

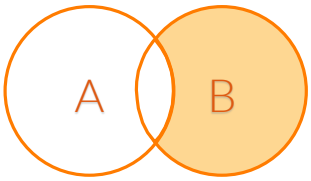
Code

```
SELECT B.ID, A.[First Name]
, B.[Last Name]
FROM [First] A
RIGHT OUTER JOIN [Last] B
    ON A.ID = B.ID
WHERE A.ID IS NULL
```

or

```
SELECT B.ID, A.[First Name]
, B.[Last Name]
FROM [First] A
RIGHT JOIN [Last] B
    ON A.ID = B.ID
WHERE A.ID IS NULL
```

Logic



Connection

NULL ⟷ B

Joins: FULL OUTER JOIN

A FULL OUTER JOIN between 2 tables returns all records from both tables including their intersection.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

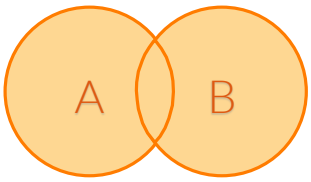
Result

ID	First Name	Last Name
A1	John	Smith
A2	Samantha	*NULL*
A3	Paul	Johnson
A4	*NULL*	Adler

Code

```
SELECT CASE WHEN A.ID IS NULL
            THEN B.ID
            ELSE A.ID
       END as ID
FROM [First] A
FULL OUTER JOIN [Last] B
    ON A.ID = B.ID
```

Logic



Connection

A ⟷ B
A ⟷ NULL
NULL ⟷ B

Joins: FULL OUTER JOIN (exclude)

A `FULL OUTER JOIN` with exclusion between 2 tables returns only records from both tables excluding their intersection.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

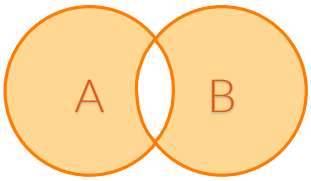
Result

ID	First Name	Last Name
A2	Samantha	*NULL*
A4	*NULL*	Adler

Code

```
SELECT CASE WHEN A.ID IS NULL
            THEN B.ID
            ELSE A.ID
       END as ID
, A.[First Name] , B.[Last Name]
FROM [First] A
FULL OUTER JOIN [Last] B
    ON A.ID = B.ID
WHERE A.ID is NULL
    OR B.ID IS NULL
```

Logic



Connection

A ⟷ NULL
NULL ⟷ B

Joins: CROSS JOIN

A `CROSS JOIN` between 2 tables returns every combination of records from one table to the other.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

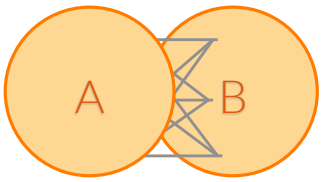
Result

First Name	Last Name
John	Smith
Samantha	Smith
Paul	Smith
John	Johnson
Samantha	Johnson
Paul	Johnson
John	Adler
Samantha	Adler
Paul	Adler

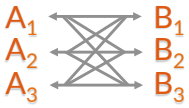
Code

```
SELECT CASE WHEN A.ID IS NULL
            THEN B.ID
            ELSE A.ID
SELECT A.[First Name]
, B.[Last Name]
FROM [First] A
CROSS JOIN [Last] B
```

Logic



Connection



Matching foreign keys in tables

familydb.py

```
foreign_keys = [
  {
   'tablename':'family_member',
   'fks':[
           {
            'name': 'fk_family',
            'query': """
              ALTER TABLE family_member
                ADD CONSTRAINT fk_family
                FOREIGN KEY (family_id)
                    REFERENCES family(id)
                    ON DELETE CASCADE
              """
           }
  ]}
,
```

1. Name of the foreign key
2. Sets up the query
3. Specifies which table to alter
4. Identify the name of the constraint
5. Identify the column for foreign key
6. Identify the table (column) to refer to

Tells SQL to delete entries from the child table if they're deleted from the parent table

SQL

```
ALTER TABLE family_member
 ADD CONSTRAINT fk_family
    FOREIGN KEY (family_id)
        REFERENCES family(id)
        ON DELETE CASCADE
;
```

Matching foreign keys in tables

familydb.py

```
# ...Continued from the previous page...
{
  'tablename': 'family_member_interest',
  'fks':[
          {
           'name': 'fk_pi_family_member',
           'query':"""
              ALTER TABLE family_member_interest
                 ADD CONSTRAINT fk_pi_family_member
                FOREIGN KEY (family_member_id)
                    REFERENCES family_member(id)
                    ON DELETE CASCADE
            """
            },
```

1. Name of the foreign key
2. Sets up the query
3. Specifies which table to alter
4. Identify the name of the constraint
5. Identify the column for foreign key
6. Identify the table (column) to refer to

Tells SQL to delete entries from the child table if they're deleted from the parent table

Matching foreign keys in tables

familydb.py

```
# ...Continued from the previous page...
          {
            'name': 'fk_pi_interest',
            'query': """
                ALTER TABLE family_member_interest
                ADD CONSTRAINT fk_pi_interest
                 FOREIGN KEY (interest_id)
                     REFERENCES interest(id)
                     ON DELETE CASCADE
               """
          },
```

1. Name of the foreign key
2. Sets up the query
3. Specifies which table to alter
4. Identify the name of the constraint
5. Identify the column for foreign key
6. Identify the table (column) to refer to

Tells SQL to delete entries from the child table if they're deleted from the parent table

Defining functions

familydb.py

```
# Use 'def' to create a new function - here, we'll call the function
# 'fk_exists', and it will have the named arguments 'tbl_name' and
# 'fk_name', which will be the variables later on in the function.
def fk_exists(tbl_name, fk_name):
    cur.execute("""select * from information_schema.table_constraints
                where table_schema='familydb'
                and table_name='{}'
                and constraint_name = '{}'
                """.format(tbl_name, fk_name))
    return len(cur.fetchall()) > 0

def index_exists(tbl_name, index_name):
    cur.execute("""select * from information_schema.statistics
                where table_schema='familydb'
                and table_name='{}'
                and index_name = '{}'
                """.format(tbl_name, index_name))
    return len(cur.fetchall()) > 0
```

1. { } creates dictionaries with key-value pairs
2. fetchall returns all the rows for the current query

Dropping foreign constraints

familydb.py

```
# We'll run the functions we defined earlier to drop foreign keys
# if they exist.
for tbl in foreign_keys:
    tbl_name = tbl['tablename']
    fks = tbl['fks']

    for fk in fks:
        drop_fk = 'ALTER TABLE {} DROP FOREIGN KEY {}'
        drop_index = 'ALTER TABLE {} DROP INDEX {}'
        key_name = fk['name']

        if fk_exists(tbl_name, key_name):
            cur.execute(drop_fk.format(tbl_name, key_name))

        if index_exists(tbl_name, key_name):
            cur.execute(drop_index.format(tbl_name, key_name))

dbconn.commit()
```

Recreating the tables

familydb.py

```
for table in tables:
    print('dropping '+table['name'])
    cur.execute('drop table if exists '+table['name'])
dbconn.commit()

for table in tables:
    print('creating '+table['name'])
    cur.execute(table['query'])
dbconn.commit()


for tbl in foreign_keys:
    tbl_name = tbl['tablename']
    fks = tbl['fks']
    for fk in fks:
        cur.execute(fk['query'])

dbconn.commit()
```


Loading files into a database

familydb.py

```
# First, create a function to load in the csv files.
def load_file(tablename):
    f = open('data/'+tablename+'.csv', 'r')  Open the filepath for the csv file
    header = f.readline()

    query = """
    insert into {}({})
     values({})
    """

    line = f.readline().strip()
    n = 0
    while line:
        n += 1
        line_parsed = ','.join(["'"+val+"'" for val in line.split(',')])
        cur.execute(query.format(tablename, header, line_parsed))
        line = f.readline().strip()
    print('loaded {} lines to {}'.format(n, tablename))
    return n
```

Loading files into a database

familydb.py

```
# Now, use the function we defined above to load in the file names.
n_families = load_file('family')
n_fam_members = load_file('family_member')
n_interests = load_file('interest')
dbconn.commit()
```

family.csv

	A
1	last_name
2	williams
3	johnson
4	myers
5	myers
6	robinson

family_member.csv

	A	B	C	D
1	first_name	birth_date	family_id	relation
2	mark	19650316	1	f
3	francesca	19641118	1	m
4	thomas	19901209	1	c
5	mary	19920821	1	c
6	helen	19941014	1	c
7	thomas	19821207	2	f
8	catherine	19830813	2	m
9	anthony	20100415	2	c
10	patricia	20120517	2	c

interest.csv

	A
1	interest
2	cooking
3	sports
4	running
5	reading
6	travel
7	biking
8	comics
9	music
10	history
11	programming
12	film

Updating ages

familydb.py

```
# Update the current_age field to calculate their age as of today.
# We'll need the DATEDIFF function, and the MySQL internal CURRENT_DATE variable
cur.execute("update family_member
             set current_age = floor( DATEDIFF(CURRENT_DATE, birth_date)/365) ")
dbconn.commit()
```

Creating people's interests

familydb.py

```
# First import numpy and set the seed to make it reproducible
import numpy as np
np.random.seed(100)

# Next, set the values. The random.randint function returns a random integer. The
# arguments define the parameters.
n_person_interests  = n_fam_members*3
rand_person_ids = np.random.randint(1, n_fam_members+1, n_person_interests)
rand_interest_ids = np.random.randint(1, n_interests, n_person_interests)

for i in range(n_person_interests):
    query = """
    insert into family_member_interest(family_member_id, interest_id)
    values({},'{}')
    """.format(rand_person_ids[i], rand_interest_ids[i])
    cur.execute(query)
dbconn.commit()

print('loaded {} lines to family_member_interest'.format(n_person_interests))
```