Using SQL With Python

In this lecture we will learn how to work with a SQL database using Python. Today we will cover the following sections:

- Getting a database up and running with MAMP and phpMyAdmin
- Connecting your database into a Python script
- SQL basics using a small sample data set
- Using an ORM with a sizeable data set

MAMP and phpMyAdmin

Intro

We will be using MySQL as our database backend. MySQL requires a database server in order to use it. You can either download the server software directly from the MySQL Website, or through a third party such as MAMP. We have suggested the use of MAMP since it budles several useful programs together, including phpMyAdmin, which provides a lightweight database front end, and Apache web server, which allows us to use our computer to host a local website.

To begin, launch MAMP and if they do not start automatically, click the "Start Servers" button. In a few seconds, you should see green lights next to both Apache and MySQL server.

Now click the "Open start page" button and click on Tools > phpMyAdmin. Alternatively, go directly to `http://localhost/phpMyAdmin` in your internet browser.

Now we'll have to give ourselves a username on our local server (`localhost`). Screenshots of the following steps can be found in the `phpmyadmin-create-user` folder of your lecture materials. In the top menu bar, click "User accounts." About halfway down on the page that opens will be a link to "Add user account." Fill out the login information (top panel):

- User name - Create a user name for yourself
- Host name – Choose "Local"
- Password - Optional to create a password. Leave blank to skip

In the Global privileges section (third panel), you can click "Check all" to grant yourself all privileges to databases on localhost.

The `SSL` option in the following panel can be `REQUIRE NONE`.

Finally, to create the user click the "Go" button in the bottom right. Now we can begin using Python to connect to our database server!

Creating a database

Now we'll create a database called `familydb`. Still in phpMyAdmin, at the sidebar on the left, simply click on the "+New" button to create a new database. Enter "familydb" as the name for the database and click "Create". You do not need to create any tables; this will be done in the next step.

Now that you have created a username and optional password, open the `familydb.py` script from Spyder, making sure to substitue your own username in the connection string at the top:

```
# connect to the db
import mysql.connector
dbconn = mysql.connector.connect(user='ta_anna', # substitute your user name here
                                 password='', # leave password blank if you
didn't create one
                                 host='127.0.0.1',
                                 database='familydb')
cur = dbconn.cursor()
```

Make sure your Spyder working directory is set to the `family` folder. Now run `familydb.py`, which will create and populate the tables with some sample data.

In most cases, you will be working with an existing database, so you will not be populating it yourself. However, if you are working on a research project, you will have to have a way to load the data. Most database GUI's (such as phpMyAdmin and MySQL Workbench) have a window interface to load a table from a file, but this will generally be too tedious for your needs, and a code solution will be preffered.

There are many options for loading data, ranging from as simple as bulk loading a file from the command line, to using what is called an *object relational mapper (ORM)* to parse and clean each value before adding it to the database. In our first example we will not make use of an ORM, but later we will see how to use one of the most popular ORM's: SQLAlchemy.

`mysql` Python library

We would like to work with a MySQL database from a Python script. This could be useful if you're writing a script that reads or writes data from a database. We will see another use for connecting a database to your Python script in the next lecture where we learn how to make a basic web application with Python.

The documentation for the MySQL connector library can be found here.

To import the MySQL connector library, we write:

```
import mysql.connector
```

Then to make a connection to an existing database, we write:

```
dbconn = mysql.connector.connect(user='MYUSERNAME',
                                 password='', # leave password blank if you
didn't create one
                                 host='127.0.0.1',
                                 database='familydb') # all of the queries we'll
do will use this database
```

Notice that the `dbconn` variable has type `MySQLConnection`. 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()
```

As an alternative, you can also say `cur.fetchone()` to select just the first result, or `cur.fetchmany(n)`, where for `n` you put the number of records you want.

Note that 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

In addition to retrieving data, we can also run database operations through `cur.execute()`. These do not return rows, but rather change something about the database such as adding or modifying a table. However, any changes to the database must be *committed* in order to take hold. For example we could write:

```
cur.execute('DROP TABLE family')
```

This would run the drop command, but the table drop would be in a "pending" phase until we commit it:

```
dbconn.commit() # note the commit() is a method of dbconn not cur
```

Only after the commit is the table permanently dropped. If you do not commit, the changes will not be saved, and upon ending your session, the changes will be "rolled back", ie., everything will be reverted to before the change. You can run multiple commands before committing. This way if any one of them hits an error, none of them will have been committed.

SQL Basics

Relational vs Non-Relational databases

A relational database, which is the kind of database we will be working with, consists of various tables that have explicit relationships to each other. The design of the tables has historically followed a standard of using the least disk space as possible. However, this can lead to complicated and sometimes computationally intensive queries. More recently, non-relational databases (so called NoSQL "not only SQL"), have become popular. These store the data in a less efficient way (worse for disk space usage), but at the benefit of faster and more simple queries. As disk space has become quite cheap, this is an appealing tradeoff. In our course, we will still concern ourselves with relational database, since their use is widespread.

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. Generally speaking, all of these use the same basic query syntax. However they will typically differ on specific functions. 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. Additionally, different management systems will require different Python libraries to incorporate with your code. We will be using the MySQL Connector library for Python. If instead you were working with a PostgreSQL database, for example, you would use the `psycopg2` Python library. It's just a matter of researching what library is used.

Tables and Data Types

Each table in a MySQL database is defined as follows:

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

We define the table name, and the columns that it has. Each column must be given a data type. By default, columns are allowed to have missing (null) values. If we want to disallow missing values from a column, specify the `NOT NULL` keyword next to a column.

It is common practice for tables to have a primary key field defined. This can be any column or columns that make each row of the table unique. Or we can simply have MySQL auto-generate an incrementing number starting from 1. In this case, we create an `id` column for the family, and specify `AUTO_INCREMENT` which instructs MySQL to automatically create an incrementing id.

The "full address" (or *fully qualified name*) of this table is `familydb.family`: you have created the `family` table in the `familydb` database. As long as you are working within the `familydb` database, you will not need to write the full name. Just be aware of what this notation means because you will occasionally come across it.

Adding or Dropping columns

To add or drop columns from an existing table is as follows. To add a new column:

```
ALTER TABLE family_member ADD COLUMN birthday_month INT NULL
```

This adds a column called birthday_month to the `family_member` table, with type of integer and allowing nulls.

To drop a column is similar:

```
ALTER TABLE family_member DROP COLUMN birthday_month
```

Foreign Key relations

The main feature of relational databases is to store records that are related in some way. In our example we have a `family` table and a `family_member` table. Each family_member belongs to a family. We could specify each family_member's last name in the `family_member` table, but it will save us storage space in the database to simply store a reference to the family table.

We create the `family_member` table and then add a foreign key constraint to it. The foreign key not only symbolizes the relationship between the tables, it also enforces it: if we try to add a record to `family_member` using a `family_id` that does not exist in `family`, the database will block us from doing so. This means every person in `family_member` is guaranteed to have a corresponding family. The syntax for creating a foreign key is shown below:

```
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,
    family_id INT NOT NULL,
);

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

The `family_id` column will contain the id from `family` of the family_member's family. This way we don't have to store the last name in this table, which takes more space than storing just the id. We explicitely define the relationship between family_member and family by creating a *foreign key* on the `family_member` table. We can choose a name to refer to the foreign key; in this case we called it `fk_family`. This specifies which columns establish the link: the `family_id` of `family_member`, and the `id` column of `family`. The final keyword `ON DELETE` instructs MySQL what to do if someone's family is removed from the `family` table. Setting this option to `CASCADE` means that if a family is removed from the `family` table, we want to "cascade" this removal to all children, in other words, remove all corresponding family members from the `family_member` table.

> *Note: when creating and dropping tables, it is important to keep in mind any existing foreign keys. You cannot create the `family_member` table before you create the `family` table, because the foreign key declaration will fail to find the `family` table. Additionally, if you want to drop the `family` table, but keep the `family_member` table, first drop the foreign key on `family_member`. Otherwise MySQL will not let you drop a table that is related to an existing table. To drop a foreign key use the following syntax:*
>
> ```
> ALTER TABLE family_member DROP FOREIGN KEY fk_family;
> ```

Relationship types

The example of family and family_member is called a one-to-many relationship. For any *one* family, there are *many* people (family members). Note that "many" means 0, 1, or more.

Our database has an example of a *many-to-many* relationship as well: the relationship between `family_member` and `interest`. *One* family_member can have *many* interests, and *one* interest can be had by *many* people. In a relational database, many-to-many relationships are handled through an intermediate table. In our case, this is the `family_member_interest` table: both the `family_member` and `interest` tables have *one*-to-many relationship to the intermediate table.

In your lecture materials you will find a diagram of the database in `family-model.pdf`. This is called an *entity-relation* diagram. It shows each table and it's relationship to the others. One to many relationships are shown with a line that has one slash on the "one" side, and a "crows foot" on the "many" side.

Inserting data to a table

To add rows to our family table by hand, we could write:

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

For each column that we specify -- just `last_name` in this case -- we provide a value. Note that we do not have to specify the `id` column. For every column that we do not specify, MySQL will place either a `NULL` value, or if a default value is defined, the default. In our case, the default for `id` is the auto incrementing id value, so the above statement will create a new row with the id being the next value in the sequence.

Creating tables from a query

It will often be useful to create a table and populate it with the result of a query. To create a new table from our `family_member` table, we write:

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

This will create the columns of the new table `family_member_first_name` based on the columns selected from the query. However, only the values from the table will be copied into the new table; the new table will not have, for example, the same foreign keys, indexes, or column default values defined -- you would have to add this next.

Temporary tables

It will be often useful to create temporary tables to store intermediate results of a complex calculation. The syntax is the same as creating a regular table, only with the additional `TEMPORARY` keyword. Typically temporary tables will be created as a result of a query:

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

The lifetime of a temporary table is your current connection to the database, which is called your database *session*. Once your disconnect, any temporary tables are automatically dropped.

Truncating and Dropping tables

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

```
TRUNCATE family_member_interest
```

To remove the table entirely, use the `DROP TABLE` command:

```
DROP TABLE family_member_interest
```

It is not necessary to truncate a table before dropping it; you can drop the table right away.

If you run the above command a second time, you will receive an error from MySQL, since the table no longer exists, so it cannot be dropped again. Often you have a script it which depending on when it's run, the table may or may not exist, but you want to ensure the table is dropped. Then the following syntax comes in handy:

```
DROP TABLE IF EXISTS family_member_interest
```

The query language

SQL statements make use of surprisingly few keywords. This section and the ones that follow outline these commands.

The basic `SELECT` statement

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

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

Note that 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 in your table. To make sure the rows are returned in a particular order, specify an `ORDER BY` condition.

The below query selects the youngest five members of the `family_member` table. To select all columns you can use a `*` instead of explicitely listing each column name.

```
SELECT *
 FROM family_member
ORDER BY current_age
LIMIT 5
```

`WHERE` clause logical conditions

Here are some of the most common logical statements used in SQL `WHERE` clauses:

```
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)
```

Just like in Python, logical statements can be combined with `AND` or `OR`. An example might be:

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

More complicated conditions can be applied with parentheses:

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

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. The `SET` clause instructs which column to modify and the new value to give it. To specify which values to modify, we use as `WHERE` clause just as in selecting data. In the below example, we turn all of the `m`'s in the relation field into `mother`:

```
UPDATE family_member
SET relation = 'mother'
WHERE relation = 'm' /* what happens if WHERE clause not present here? */
```

We can also have set `SET` use a calculation based on another column. For example, if we create a new column for the birthday month:

```
ALTER TABLE family_member ADD COLUMN birthday_month INT NULL
```

we can populate it based on the `birth_date` field using `UPDATE`:

```
UPDATE family_member
SET
```

Deleting specific rows from a table

We saw earlier how to remove all of the rows in a table, and how to drop the table completely. If we just want remove a few rows from the table, we can use `DELETE`. To specify the rows to be deleted, we can use the same syntax for `SELECT`ing specific rows:

```
DELETE FROM interest
WHERE interest = 'reading' /* removes reading from the interest table */
```

`case` statements

To derive a new column based on the values of other columns, we can use a `case` statement:

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

Notice how we provide a name for the derived column using `AS age_type`. This is called *aliasing* a column. Any column can be aliased in this way, even those that are not derived.

Case statements can also be used to provide a custom sort order. If we want the `relation` field to be sorted `m`, `f`, `c`, instead of the default alphabetical sorting, we can use a case statement inside the `ORDER BY` clause

```
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
```

If you simply used `order by family_id, relation`, the relations would be sorted in alphabetical order (`c`, `f`, `m`).

> *Notice that here we have used the more compact alternative syntax for the `case` statement. When you are just testing different possible values of a single variable, you can use this syntax. If the logic is more complicated, use the more general syntax from the previous example.*

Go to Exercise: `sql-basics/exercise-1`

Aggregation

Oftentimes, a column will have fewer distinct values than the total number of rows. It is then possible to summarize by this column. If we take the contents of our `family_member` table, we see that there are fewer distinct values of the `family_id` (5 values) spread over all 22 rows:

```
SELECT *
FROM family_member
order by family_id, birth_date
```

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

id	first_name	relation	birth_date	current_age	family_id
21	samantha	c	1994-12-12	22	5
22	kimberly	c	1996-11-15	20	5
20	margaret	c	1999-06-19	18	5

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:

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

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

Note the `GROUP BY` keyword is crucial. This tells MySQL how to define the groups. If you leave this out, the query will run, but the result will not be what you expect.

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

```
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
```

A complete list of MySQL aggregate functions can be found here.

We 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`.

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

Or we could ask, how many people have birthdays in each month?

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

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

> *Notice that only the months that appear in the data will show up in this query. For example, there are no February birthdays, so there is no row in the results for February. We do **not** see a row with (February, 0). It is important to remember this phenomenon because you will likely come across it fairly often.*

What does the * mean in `count(*)`?

Count, like the other aggregation functions, can be given a column as an argument. However, be aware that `count(last_name)`, for example, counts only the *non null* values of `last_name`. If a group has 4 rows, but in one of them the `last_name` field is null, `count(last_name)` will return 3, which may or may not be the behavior you need. To count event non-null values, you could either use `count(x)` where the coumn `x` is a column you know won't contain nulls. However the best way to handle this is to say `count(*)`. This counts all of the *rows* in the group, as opposed to values in a particular column. This may not seem impotant now, but it will useful when we learn about outer joins.

Aggregation with a `WHERE` clause

We might ask: "How many members of each family are over age 30?" This can be answered with:

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

Filtering on an aggregated value with `HAVING`

A `WHERE` clause applies a filter to each row of the original data. What if we want to filter after we have computed the aggregate? For example, we may want to know which months have at least 2 birthdays in them. We can add a `HAVING` statement after our `GROUP BY` to filter based on the aggregate value (the count):

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

Aggregation trick using case statements

Extending the sample from above, what if we wanted to answer multiple similar questions with only a single query? For example,

1. How many members of each family are over age 30?

2. How many children in each family are older than 4?
3. How many members of each family have a birthday in December?

We could write three separate queries like the one in the previous section. However there is a trick we can use taking advantage of `case` statements that allows us to answer all three with a single query:

```
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

Go to Exercise: `sql-basics/exercise-2`

Union

To combine two or more select statements "by rows" we can use the `UNION` keyword. The select statements must contain the same number of columns, and each corresponding column must have the same data type. The column must be specified in the same order: MySQL will not make an attempt to match columns names together. The column names and data types will be taken as those from the first query.

Let's create an example with our `family_member` table.

```
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.*

To repeat: **It is extremely important that the column names match up**. The following query will run because the number of columns is the same and the corresponding data types are the same. But the output will not be what you expect:

```
/* examplel of incorrect UNION query - column misaligned in second query*/
SELECT first_name, birth_date, relation
FROM family_member
WHERE relation = 'm'

UNION

SELECT relation, birth_date, first_name /* different column ordering */
FROM family_member
WHERE relation = 'f'
```

Joins

To combine record sets "by columns", we make use of the SQL Join. Suppose we want to display each person with their first and last name. These are stored in different tables: the `family_member` table has the first name, and is linked to the `family` table via the `family_id` column. To get both the first and last name side by side, we join the tables on the family id:

```
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
```

It is common to alias the table names (as in above, `family_member` is aliased as table "a", and `family` is aliased as table "b"). Then when we need to refer to a column name from one of the tables, we can use the shorthand `a.family_id`, etc. Without the alias, you would need to say `family_member.id` and `family.id`, so the alias saves you some time. You can choose an alias that makes sense, for example `fm` and `f` would have been clearer than `a` and `b`.

If both tables contain a column of the same name, it is essential to qualify the column with the table name, in order to avioid ambiguity; for example both tables have a column called `id`, so we must qualify these as `a.id` and `b.id`. Only the `family_member` table has a column called `birth_date` so it is not necessary to provide the table name or alias there.

> *Note that we have chosen a useful convention for naming columns. The `id` field in `family` is named `family_id` when it appears in other tables. In the `family` table we just call it `id`, not `family_id`. This is a standard convention, however you are free to use a convention that makes the most sense to you or your organization.*

More complex join conditions

We can use joins to answer more complicated questions. As an example, suppose we wanted to show pairs of individuals where for each person, we show everyone who is younger than that person. The following example (which uses a "self join" -- joining a table to itself -- answers this question.

```
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

In the first example above, each row in `family_member` was guaranteed to have a match in the `family` table, thanks to our foreign key constraint. Often however, it might not be the case that for each value in the first table (often called the "left" table) there is a matching value in the second table (often called the "right" table).

```
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
```

Notice that person numbers 1, 5, 19, and 22 are missing from this table. **Question:** Why is this and how can you prove it?

In the case that there's no guarantee of a match in the right table, but we want to retain every value in the left table no matter what, we can use an *outer join* (as opposed to *inner join* which we just did). This will keep everything on the left table (including people 1, 5, 19, and 22), and whatever matches on the right. If there is no match on the right, we will still see the value from the left, and all of the right-hand side columsn will have avalue of `NULL`.

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

```
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 a.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 it would simply be equivalent to switching the order of the two tables and using a left join. For this reason it is hardly used.*

The left join above is very useful for aggregation. If we wanted to get a count of each person's interests we could write:

```
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
```

We use the count of the right hand side column `interest_id`, because we know it will be null in the case of no match, and `count(x)` only counts non-null values of the column `x`. Therefore we see a row with (id=1, num_interests=0), etc. Had we used an inner join instead, there would be no row containing person numbers 1, 5, 19, and 22, which may not be what we want.

We can also look at the second more complex example. Remember there are two individuals of age 52, which is the oldest age in the `family_member` table: Francesca and Mark, and Francesca is slightly older. So where is Francesca in our results? She does not appear on the left hand side, because there was no match on the right hand side for her, i.e., no one was found who is older that she.

So we can change the query to do a left join instead:

```
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
```

Now we see a row with Francesca on the left, and simply a blank (null) value on the right.

Cross join

A cross join is simply an inner join, where the `ON` condition is always true. For example:

```
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*/
```

This type of query is seldom used, but is presented her for sake of completeness.

Question: 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

We have seen joins used two combine two tables, but the inputs to a join need not be tables. You can also use a query, surrounded by parentheses, as a pseudo-table, such as in the following example:

```
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 /* "table" b is called a subquery */
ON a.id = b.family_id
```

Subqueries are extremely useful for combining information on the fly. Complicated subqueries however can become computationally expensive, and then it becomes better to create a temporary table instead.

> *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."*

Question Go through each of the four columns in the result set and be sure you know where they came from.

Multiple joins

There's no reason to limit ourselves to two joins. The following query combines `family`, `family_member`, and a subquery that counts the number of interests per member.

```
SELECT first_name, last_name, birth_date, coalesce(n, 0) as num_interests
FROM family as a
JOIN family_member as b
  ON a.id = b.family_id
LEFT JOIN (
  select family_member_id, count(*) as n
  from family_member_interest
  group by 1) as c
ON b.id = c.family_member_id
```

Note: the left join will return a null value for `n` when there is no interests for that person. To convert a null to some other value, such as 0 in our case, we use the `coalesce` function.

Question: What would we get if the third table was `family_member_interest` itself, and not this condensed subquery?

Go to Exercise: `sql-basics/exercise-3`

Indexing for better query performance

Computers are able to perform a lot of calculations very quickly. If the size of your tables is no more than a few hundred rows, you should not have to do any type of engineering to have fast running queries. However, as the size of your data grows, you will need to consider different strategies for performance. While this is a huge area, we will talk about one strategy that can greatly improve the speed of queries with `WHERE` clauses and / or joins.

First a bit of background. When you make the following query, for example:

```
SELECT *
FROM family_member
WHERE relation = 'c'
```

the database engine reads every row in the table and decides whether or not to keep it. This is called a *full table scan*, and can become very time consuming as the size of the table grows. If this is a `WHERE` clause that you need to use often, you can put an *index* on the `relation` field. Think of an index as a phonebook for the column. In the process of creating an index, the database first finds the unique values of this column (which are 'm','f', and 'c'). It then looks up which rows each of these values appear on. So next time you run a query with `WHERE relation = 'c'`, it already has the location of these values stored in its phonebook, and it doesn't need to scan every row. Note however, an index will of course take up storage space in order to hold all of this information. So the tradeoff is storage space for query speed, and most developers lean towards the latter, since space is less costly than computing power.

To create this index, use the following syntax:

```
CREATE INDEX idx_family_member__relation ON family_member(relation);
```

To drop an index the syntax is simply:

```
DROP INDEX idx_family_member__relation on family_member
```

The name `idx_family_member__relation` can be whatever we choose. Use a naming convention that makes sense to you.

By the same reasoning as above, indexing columns that are going to be frequently joined on will greatly speed up the join performance:

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

To create these indexes, use the following syntax:

```
CREATE INDEX idx_family ON family(id);
CREATE INDEX idx_family_member ON family_member(family_id);
```

> *Note: MySQL automatically puts an index on columns used in foreign keys. This is because you will be presumably be making this join often. You can drop the index at any time with DROP INDEX.*

Getting metadata from `information_schema`

In addition to any databases you may create (also called *schemas* in MySQL), there are some built-in schemas that are automatically created and maintained by MySQL. One such schema is called `information_schema`. You will not see it in your list of databases in the left-side panel of phpMyAdmin or MySQL Workbench, but you can nevertheless query from it.

Two very useful tables are `information_schema.tables` and `information_schema.columns`. Take a look at the tables you have created so far with the following query:

```
SELECT *
FROM information_schema.tables
WHERE table_schema = 'familydb'
```

Replace this with `information_schema.columns` to see the columns in each of the tables.

Question: What are all of the tables available in the `information_schema`?

*ORM's: SQL-less queries

> *This section is provided as optional, for those who are already proficient in SQL. It is not necessary to learn ORM's to complete your capstone project: your code will look a little nicer with one, but they have a steeper learning curve. It is more important that you are confident in knowing what your code is doing.*

> *Note: In this section we will interact with our databases using the `sqlalchemy` library as opposed to the `mysql.connector` from the first part of the lecture. Make sure to keep the two syntaxes straight!*

As we have seen, even some straightforward sounding questions can be decently complicated to answer with a SQL query, sometimes requiring several subqueries. In addition, unless you have a high level of comfort with the language, some queries are non-intuitive to write, and therefore could be error-prone in the hands of an inexperienced programmer.

Object Relational Mappers (ORM) exist precisely to combat these issues. An ORM allows us to do all of the SQL operations we've learned, such as creating tables and retrieving rows, while only ever writing in Python! This way, the syntax is much more readable, and database operations are approachable for someone who has experience with Python but not SQL. There are a number of ORM's but the most popular is SQLAlchemy.

To use SQLAlchemy with MySQL, we will need to activate an additional library through Anaconda. Open Anaconda Navigator, and click on the Environments tab at the left. Click on the search bar on the right-hand side and type `pymysql`. When the result appears, check the box, and then click Apply. This will lead you through the process of activating the library.

Go back to Spyder and type the following into your IPython console to make sure all the necessary libraries are available:

```
from sqlalchemy import create_engine
engine = create_engine('mysql+pymysql://ta_anna@localhost/sampledb')
```

If both of these statements work, then you are ready to begin using SQLAlchemy.

Python Classes

SQLAlchemy makes extensive use of Python Classes, so we will describe these here. A Class is no more than a template to build an object. Suppose I was maintaining a list of addresses. I might have a script that looks like this:

```
addr1 = {'street_num': 3700, 'street': 'O St NW',
            'city': 'Washington', 'state': 'DC', 'zip5': '20057'}

addr2 = {'street_num': 2400, 'street': 'Sixth St NW',
            'city': 'Washington', 'state': 'DC','zip5': '20059'}

addr3 = {'street_num': 2121, 'street': 'I St NW',
            'city': 'Washington', 'state': 'DC', 'zip5': '20052'}

addr4 = {'street_num': 4400, 'street': 'Massachusetts Ave NW',
            'city': 'Washington', 'state': 'DC', 'zip5': '20016'}

addresses = [addr1, addr2, addr3, addr4]
```

There might be different things I'd need to do with an address. I could write a function for whatever that may be:

```
def print_address(addr):
    return '{street_num} {street}\n{city}, {state} {zip5}'.format(**addr)
```

What if we dealt with addresses in many different scripts? We would always need to create these functions. Also notice that every address is a dictionary with the same key. Classes let us take advantage of the similiar structure of each address dictionary, and allow us to "bundle" functions in one place that we want to use on this type of object.

To convert this example to use a class is as follows:

```
class Address(object):
    def __init__(self, street_num, street, city, state, zip5):
        self.street_num = street_num
        self.street = street
        self.city = city
        self.state = state
        self.zip5 = zip5

    def fmt(self):
        return '{street_num} {street}\n{city}, {state}
{zip5}'.format(**self.__dict__)

addr1 = Address(3700, 'O St NW', 'Washington', 'DC', '20057')
addr1.street
print(addr1.fmt())
```

We'll go through each part one by one:

- The name of this class is Address. It is customary to use title case for class names.
- The `addr1` variable that gets created after the class definition is an *instance* of the Address class. Remember: "class" means template, and "instance" means an object that was built from that template.
- The Address class "inherits" from the `object` class. We'll get to inheritance later, but for now just know that inheriting from the `object` class (which is part of base python), means that Address will automatically have some useful features.
- The `__init__` method. Every class must contain this, since it tells python how to construct your object (hence, `__init__` is also referred to a the constructor function). The arguments are whatever the values you want the user to provide. `__init__` has a parculiar first argument: `self`. Think of self as a stand-in for any instance that might get created. For example, when we created `addr1`, the various arguments we provided, such as 3700 for `street_num`, were attached to that instance. By giving a value to `self.street_num = ...`, this means we can later write `addr1.street_num` to retrieve its street number. Any of the values assigned to `self` are called *instance attributes*.
- Both `__init__` and `fmt` are example of *methods*. There are different types of methods which we won't cover here, but just to note that these two are examples *instance methods*. Being an instance method means every instance of the class (for example `addr1`) will be able to use this method. This is why we can say `addr1.fmt()`. Notice how the `fmt` method makes use of instance attributes via `self`.
- It is required for every instance method to take `self` as the first argument. Notice though that we don't need to provide a value for `self` when we call the method. For example we don't need to say `addr1.fmt(addr1)`. There is nothing in the code above that explains why this is the case, just know that instance methods are *bound* to every instance, meaning the `self` argument is automatically populated with the instance that's calling the method.

Class inheritance

We can create a *subclass* of a class, which means we will take the existing template of the class as a starting point and then modify it somehow. Subclasses contain all of the attributes and methods of the parent class, and we can choose to modify any of them.

As an example, we'll look at the SQLAlchemy `Base` class, which will be inherited by all of the classes we will create. The `Base` class is set up to be a template for a database table. It internally stores information about connection to the database, and keeps track of what tables are in the database. In addition, it lets us create our database tables without writing any SQL.

As we mentioned, to have a class inherit another class, we specify the parent class in the class declaration:

```
# different data types need to be imported
from sqlalchemy import Column, Integer, String, Date

class Family(Base):
    __tablename__ = 'family'
    id = Column(Integer, primary_key=True)
    last_name = Column(String(50))
```

When we use SQLAlchemy, we create subclasses of the `Base` class, which is a class provided by SQLALchemy. Each of the subclasses we make will correspond to (*map*) a table in our database. For each subclass, we provide the name of the table via `__tablename__`. This particular attribute is special to `Base` and lets it know what the name of the table is. Note that there is no mention of `self` here. This means `__tablename__` is bound to the *class*, and not instances of the class. For this reasons it's called a *class attribute*.

After the `__tablename__`, we see column definitions. Note that the same information is provided about each column that we provided in the SQL `create table` statement, but the way it is provided uses familiar-looking Python syntax.

Exercise:

Complete the rest of the tables: FamilyMember, FamilyMemberInterest, and Interest. Save as a file called `model.py`. You will need to include the following lines at the top of the script:

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
engine = create_engine('mysql+pymysql://ta_anna@localhost/sampledb')
  # replace "ta_anna" with your username
Base.metadata.bind = engine
```

Querying with `SQLAlchemy`

Now that we have created an *object mapping* for each of our database tables, we can work with them as if they were regular python objects. To begin working with the database, we will need to create a session. This is similar to how we created a `cursor` object before.

```
from sqlalchemy.orm import sessionmaker
DBSession = sessionmaker(bind=engine)
session = DBSession()
```

To do a simple select statement from the `family_member` table we write:

```
q = session.query(FamilyMember)
people = q.all()

# or more compact:
people = session.query(FamilyMember).all()
```

To add a `WHERE` statement is as follows:

```
adults = session.query(FamilyMember).filter(FamilyMember.current_age >= 18).all()
```