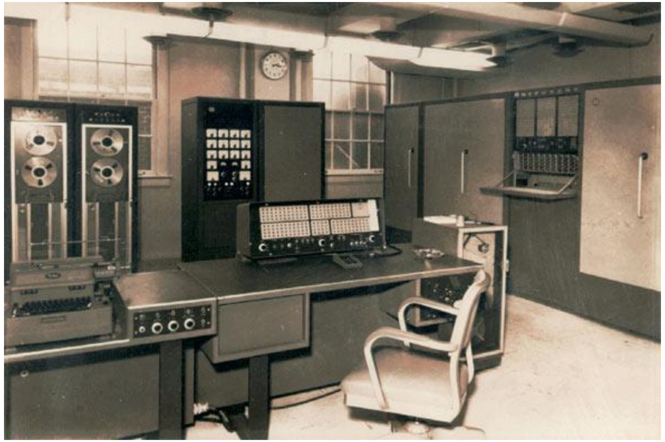
Advanced SQL II

THE KNOWLEDGE HOUSE

Agenda - Schedule

1. SQL Leetcode Q
2. Understanding Entity Relationship Diagrams
3. Using SQLite in Python
4. Break
5. SQLite Lab



Database systems of the past

Agenda - Announcements

- No pre-class quiz
- TLAB #3 due 5/14
  - Early grade due date: 5/7
  - Extension due date: 5/13
- In-class end of phase project is being released **THIS THURSDAY! (We recommend attending this review session)**

Agenda - Goals

- Interpret ERD Diagrams.
- Connect to a SQLite database using Python.
- Create a table and insert data into it.
- Run simple SQL queries from Python to explore data.
- Print and interpret results returned by the database.

SQL Leetcode Q

584. Find Customer Referee

Easy | Topics | Companies | Hint

SQL Schema > Pandas Schema >

Table: `Customer`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| name        | varchar |
| referee_id  | int     |
+-------------+---------+
In SQL, id is the primary key column for this table.
Each row of this table indicates the id of a customer, their name, and the id of the customer who referred
them.
```

Take 10 minutes to complete "Find Customer Referee": https://leetcode.com/problems/find-customer-referee/description/?envType=study-plan-v2&envId=top-sql-50

Entity Relationship Diagrams (ERDs)

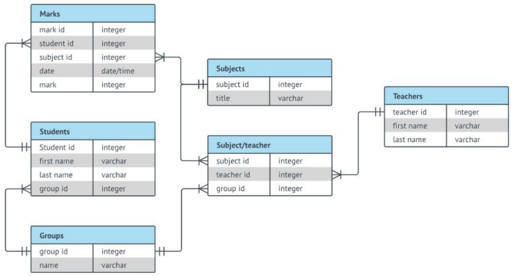
Entity Relationship Diagram (ERD)

This previous week, we learned that we often **store multiple tables** in one database.

Like we discussed, we abide by this structure due to the principles of **normalization** to ensure that we have **data integrity**.

One way to visualize the relationship of these tables is through an ***Entity Relationship Diagram***

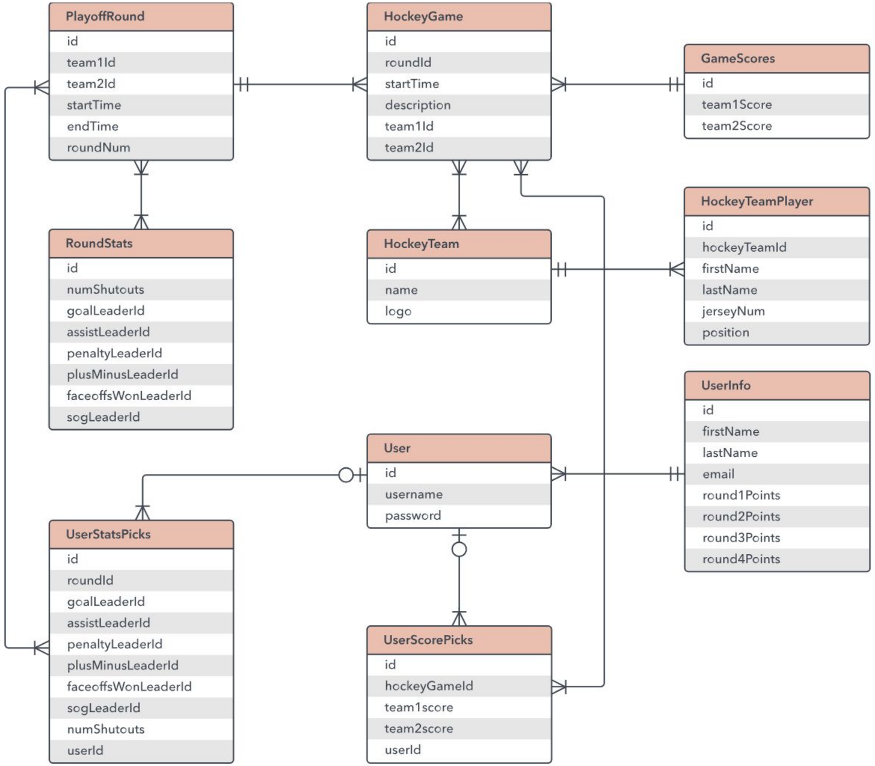
Why might we want to keep track of the structure of our database visually? What might this help us with?



What is an ERD?

An **Entity Relationship Diagram** (ERD) is a visual representation of the relationships among entities in a database. Let's iron out some definitions of an ERD:

- **Entities**: Represented by **rectangles** and are the **objects we are interested in keeping information on** (*e.g., Customer, Product, Student*). **Rows of a table.**
- **Attributes: Characteristics** of entities (*e.g., CustomerID, ProductName).* **Columns of a table.**
- **Relationships:** Connections between entities, indicating how they are related.

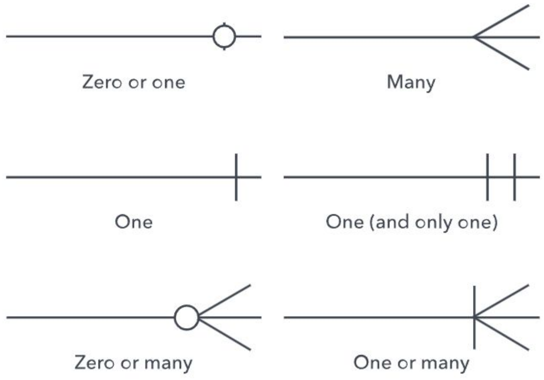


Note how attributes are listed under an entity, and we connect entities via different **types of relationships**.

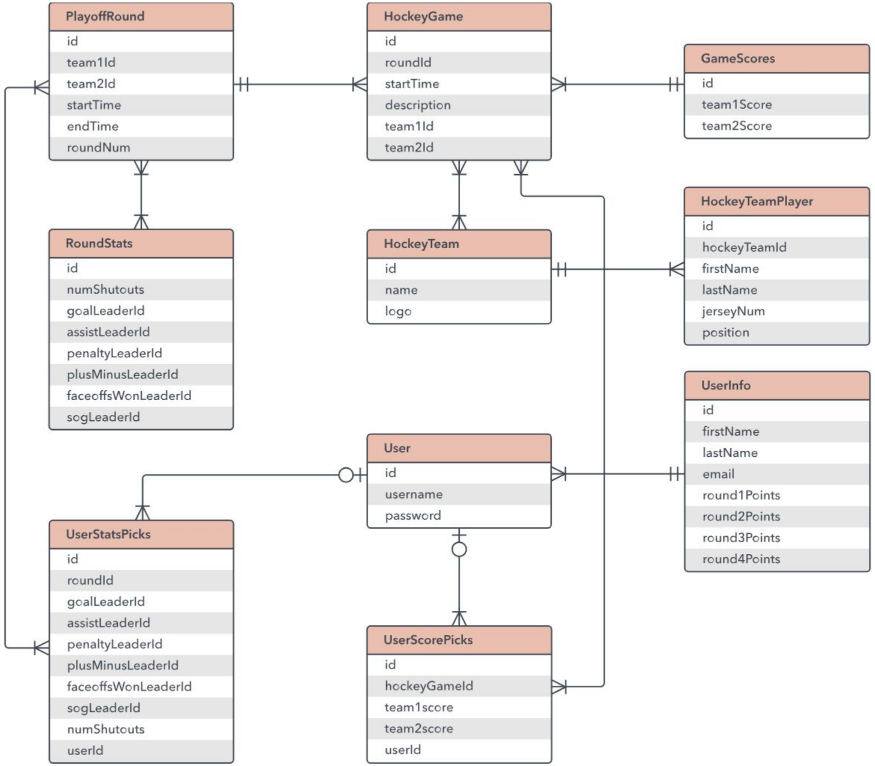
ERD Relationships/Cardinality

While **notation** for these types of relationships vary, they generally describe the following:

- **One-to-One (Zero or one)**: **One** entity is strictly associated with **one** entity (ex: *one SSN per person, and one person per SSN*)
- **One-to-Many**: **One** object can be associated with **many** other objects (ex: *one person can have multiple orders, but one order can only have one person*)
- **Many-to-Many**: **Multiple** objects can be associated with **multiple** other objects (ex: *one student can take multiple any courses, and one course can have multiple students*)



See if you could interpret the relationship between other entities, such as HockeyTeam to HockeyGame; GameScores to HockeyGame; HockeyTeam to HockeyTeamPlayer.



For example, in this diagram the **User** to **UserStatsPicks** entities is **one (or zero) to many**. In other words, one user **can have many** stats picks, but one stats picks **can only have one** user.

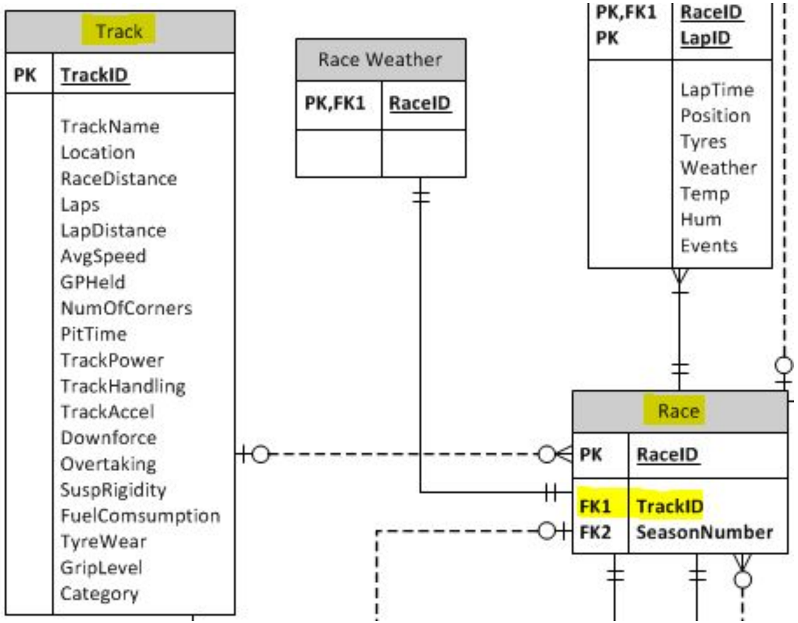
ERD Keys

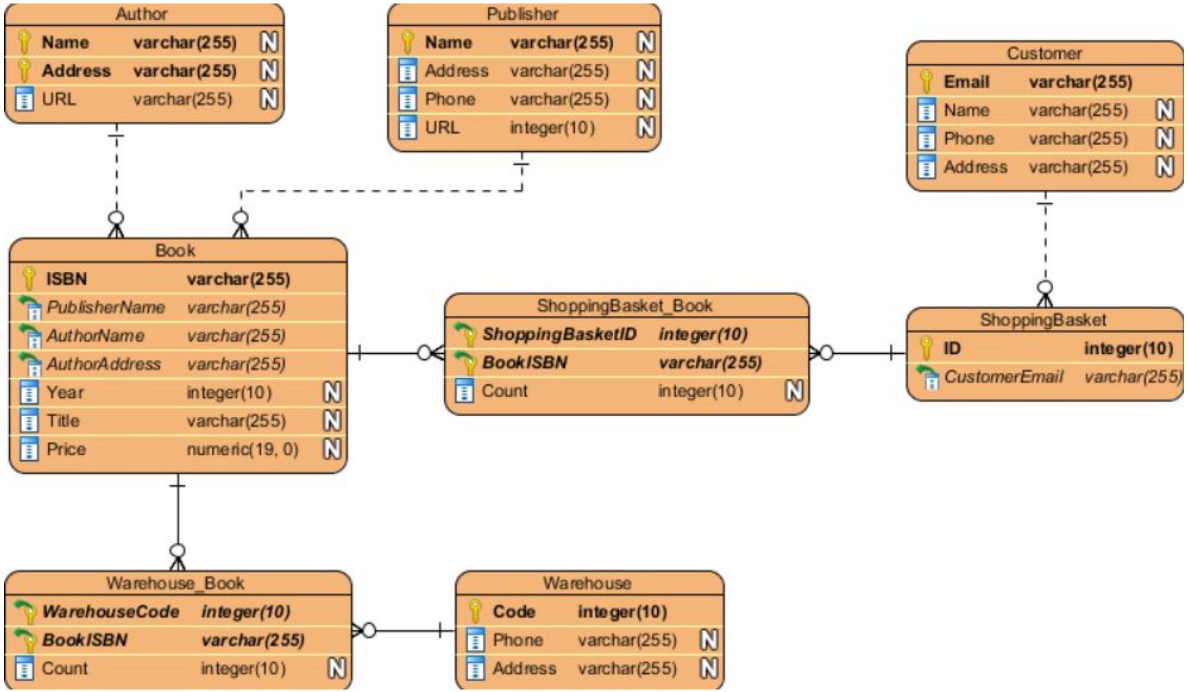
To sufficiently discuss ERD diagrams, we should also continue our discussion of **primary/foreign keys**.

Primary Keys are a unique identifier for each object in a table. It distinguishes one record from another within the same relation.

Foreign Keys are a column in a database table that is used to establish a link between two tables. It creates a relationship by **referencing the primary key of another table.**

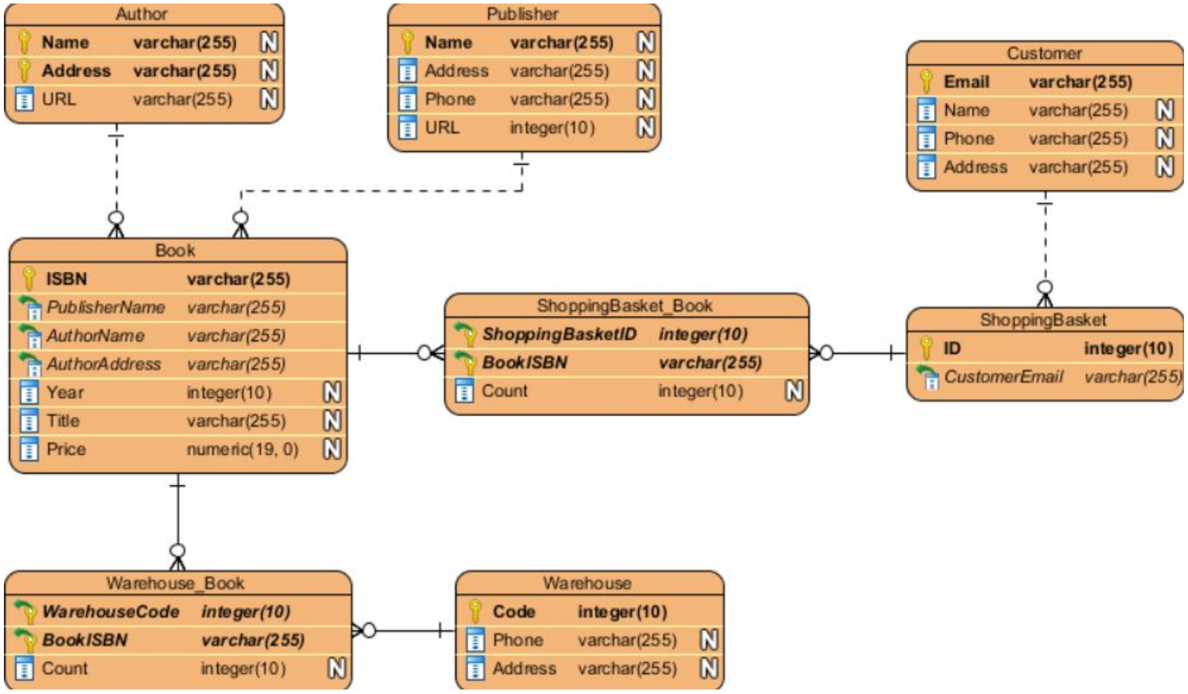
Note that TrackID shows up as a Primary Key in the "Track" table and a foreign key in the "Race" table.



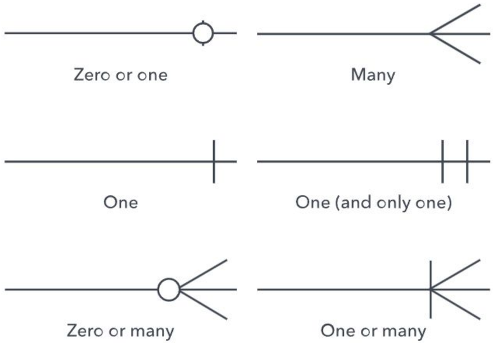


ERD Diagrams allow us to document and understand the structure of a database. Additionally it also allows us to **anticipate** the types of **JOINS** that we will perform when we are querying our database.

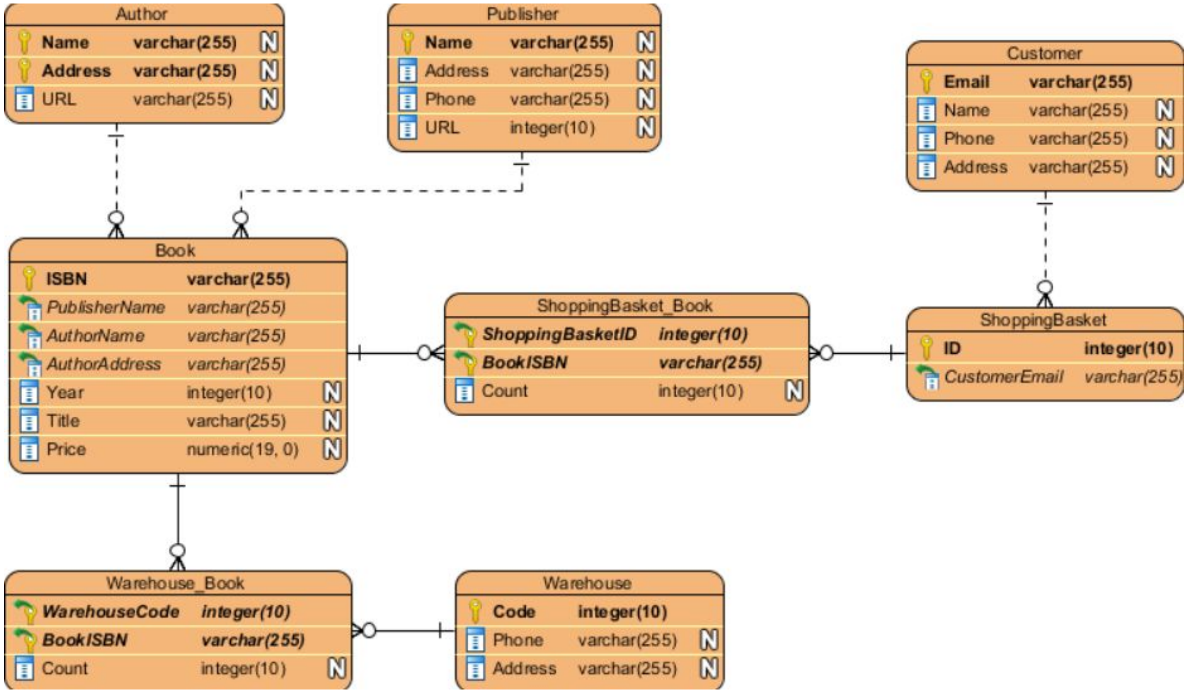
For example, let's take a look at the ERD diagram above. If we would like to get all available information on a **Book** and its **Author,** which tables should we join and on which columns should we be performing this join?




```
SELECT *
FROM Book b JOIN Author a
        ON b.AuthorName = a.Name
;
```

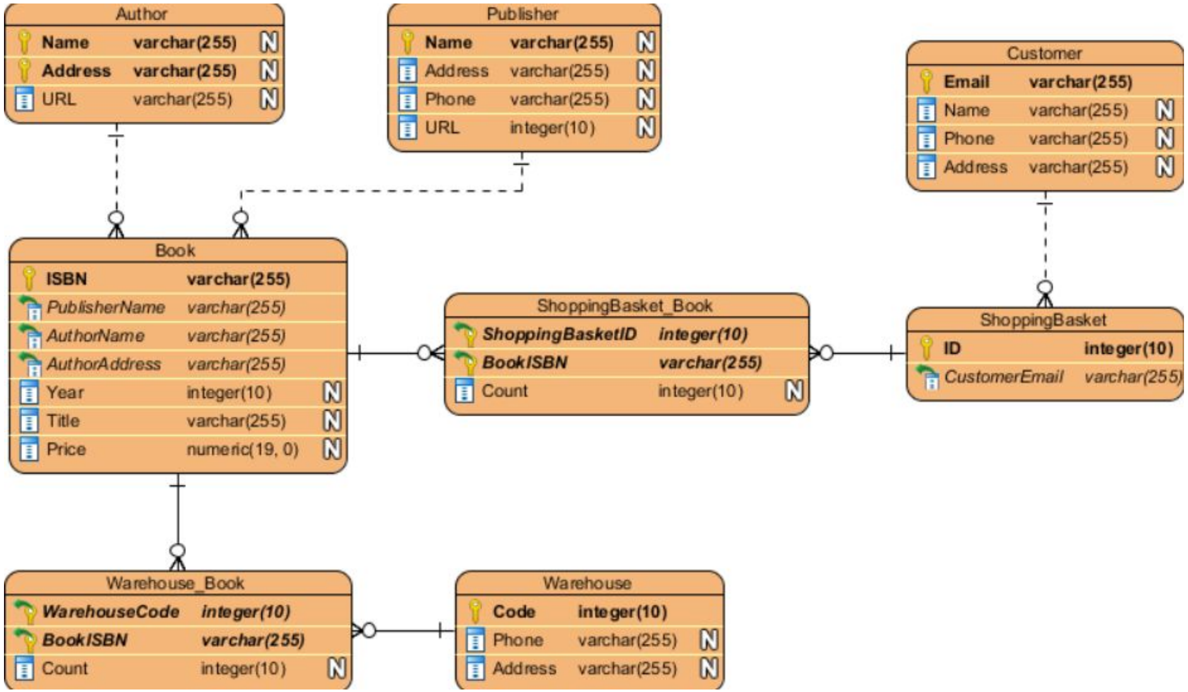


We can join the **Book** and **Author** table on the "b.AuthorName" **foreign key** and "a.Name" **primary key**. Using the ERD diagram **relationship symbols**, we can also state the relationship between these two tables. **Can one author have more than one book?**




```
SELECT *
FROM Book b JOIN Author a
        ON b.AuthorName = a.Name
;
```

Yes! But also, note that an author can have **zero books** as well. If we want to preserve authors who have zero books in this query, **what kind of join can we use?**




```
SELECT *
FROM Book b LEFT JOIN Author a
        ON b.AuthorName = a.Name
;
```

As we discussed in previous lessons, the **LEFT JOIN** ensures that we keep Authors who have not published any books (aka rows that result in null values).

Using SQLite in Python

SQLite + Python

Like we established last week, relational database management is tantamount for contemporary tech-stacks as they provide **a central system for storing & querying data.** This includes:

- ***Storage of massive amounts of data***
- ***Access via a query language***
- ***Durability even during power failures***
- ***Allow multiple users to interact with data***

But how do we actually go about implementing this technology in the context of everything we've learned so far? To accomplish this, we will use **SQLite**.

ANSI

American National Standards Institute

Like we established, there exist **different flavors** of SQL. We might opt for PostgreSQL for vector-database support, MySQL for typical LAMP applications. However SQLite is a perfect package for **non-server** database based approaches.

SQLite + Python

While almost all other SQL approaches require you to set up a database server, **SQLite** is special in the sense that is natively supported in Python and stores the database within a **lightweight local file**.

Simply import the **sqlite3** package in your Python module to get started!

Before we run through an example, let's consider if a **Python module** or **Jupyter notebook** is appropriate for this exercise…

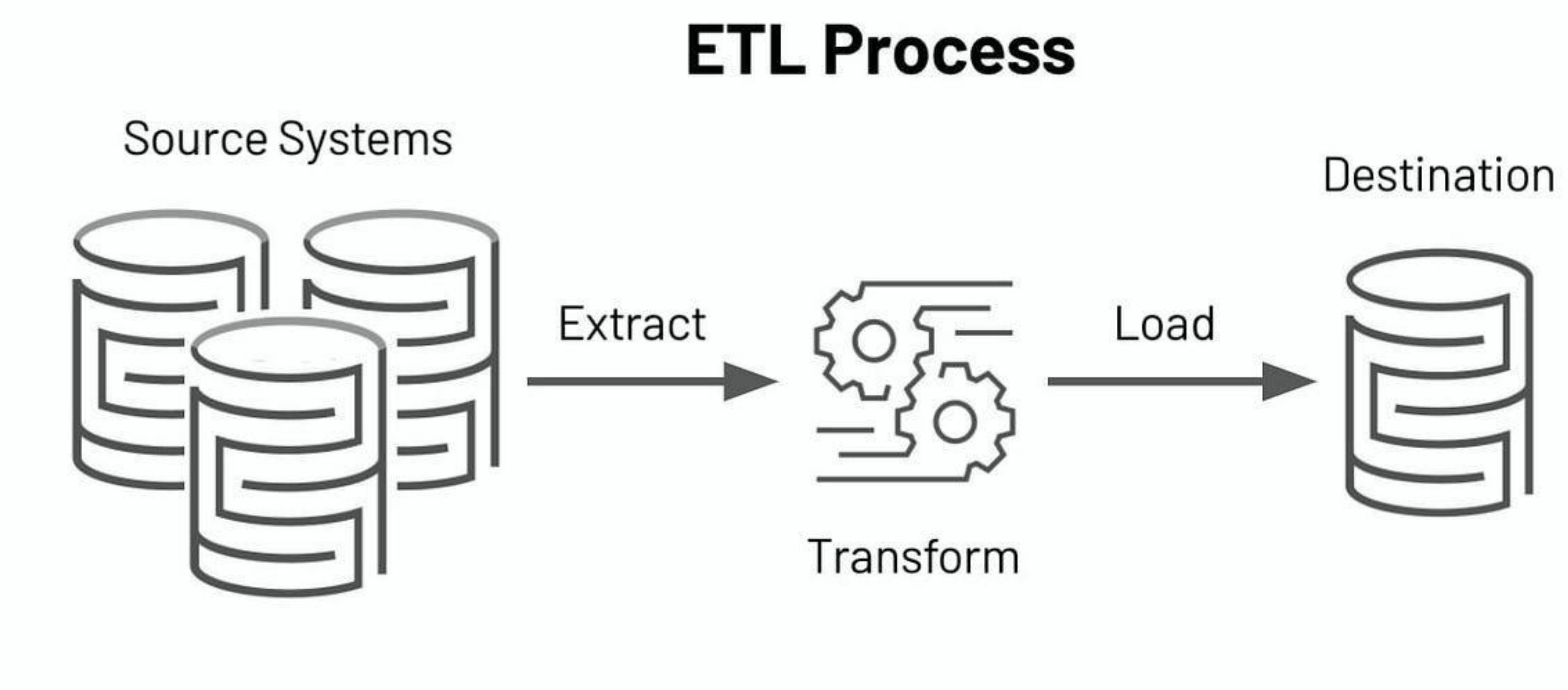
Choosing the Right Tool

Remember! Part of being a **good** technologist is being able to discern which tools are right for a specific task.

Follow this rule of thumb:

- **Python Modules:** Any functionality that should be run **automatically** (on a schedule) or **integrated into a larger system**.
- **Jupyter Notebooks**: Making **tutorials, reports, or code** that you will share with a non-technologist stakeholder.

Given these slides from "Applied Rest APIs I", which tool should we opt for in this case?



We will often integrate SQL-adjacent packages in our ETL pipelines which should be run on a **scheduled and consistent basis**. Therefore, let's utilize **Python modules** for this exercise.

```
import sqlite3

conn = sqlite3.connect("Fellowship.db")

cursor = conn.cursor()

cursor.execute(“SELECT * FROM fellows;”)
```

Within the **sqlite_intro** folder let’s access **query_db.py** to find out how we can run SQL in Python.

```python
import sqlite3

conn = sqlite3.connect("Fellowship.db")

cursor = conn.cursor()

cursor.execute("SELECT * FROM fellows;")
```

We establish our connection to the database using the **connect()** method. We supply the path to an already created database file called "Fellowship.db."

```
import sqlite3

conn = sqlite3.connect("Fellowship.db")

cursor = conn.cursor()

cursor.execute("SELECT * FROM fellows;")
```

Even though we established a connection, we still cannot read from the database until we create a **cursor**. This object allows us to **execute** SQL statements and **fetch** results from the database itself.

```python
import sqlite3

conn = sqlite3.connect("Fellowship.db")

cursor = conn.cursor()

cursor.execute(“SELECT * FROM fellows;”)
```

Additionally note that you must be confident that your SQL statement is correct before writing out your query. This adds an additional challenge to our development.

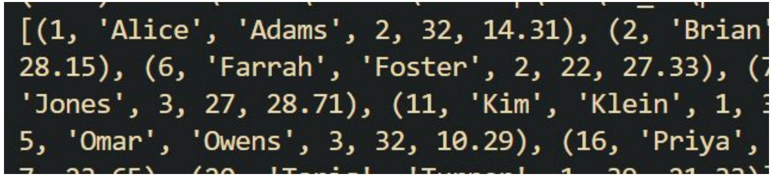
Finally, we can begin **sending** SQL statements in the form of a string to our database via the **execute()** method. Note that this requires a bit of context-switching (sort of like inserting Russian statements in the middle of an essay written in English). This is **generally considered a bad-practice**, but we will opt for this for lack of better tools (for now).

```
cursor.execute("""
    SELECT *
    FROM fellows;
""")

rows = cursor.fetchall()
print(rows)
```

We can make our SQL statements look like idiomatic SQL by introducing **triple-quoted strings**. This allows us to write strings across multiple lines.

```
cursor.execute("""
    SELECT *
    FROM fellows;
""")
```

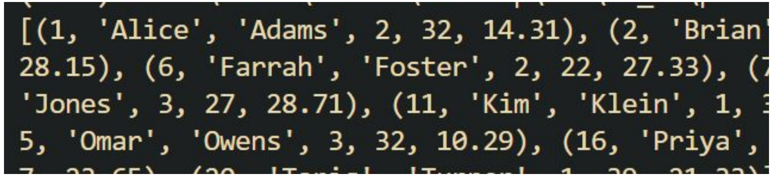



```
rows = cursor.fetchall()
print(rows)
```

More importantly however, let's focus on the output that is provide from our the statement that we just executed. Note that we get the **entire output** using the **fetchall()** method. We can have the result of this method call in a variable and print it out. **What kind of data-structure does this provide?**

```
cursor.execute("""
    SELECT *
    FROM fellows;
""")
```

```
rows = cursor.fetchall()
print(rows)
```



Note that we have a **list of tuples**! Printing out the entire list is insufficient to properly view the data. If we wanted to **iterate** through each element of the list, and **print out each tuple on a separate line**, what kind of Python syntax can we write?

```
cursor.execute(“””
    SELECT *
    FROM fellows;
“””)

rows = cursor.fetchall()
for r in rows:
    print(r)
```

```
(1, 'Alice', 'Adams', 2, 32, 14.31)
(2, 'Brian', 'Brown', 2, 27, 22.26)
(3, 'Carmen', 'Chen', 1, 38, 18.04)
(4, 'David', 'Diaz', 1, 21, 5.19)
(5, 'Ella', 'Evans', 2, 31, 28.15)
(6, 'Farrah', 'Foster', 2, 22, 27.33)
```

Since we save our list to a variable, we could instead iterate through this list using a for loop to cleanly print out our data!

```
cursor.execute(“””
    ...
“””)

rows = cursor.fetchall()
print(rows)
```

Now that we understand the syntax of running SQL statements in Python, let’s try a few exercises out! **How can we output all fellows who spent less than 10 hours on Canvas using a SQL statement? Look to the triple-quoted string in the Python module above to find out more about the columns.**

Making Changes to a DB in Python

Making Changes

While we went over running simple DML queries using sqlite3, how can we effectively **commit** a change to our database from Python?

Before we go over this syntax, let's recall the types of DDL queries that we can make, starting with the **CREATE** clause.

Statement	Description
CREATE	Create a new object in the database, such as a table or a view.
ALTER	Modify the structure of an object. For instance, altering a table to add a new column.
DROP	Remove an object from the database.
RENAME	Rename an existing object.

Taken from the [Azure DP-900 prep]

SQL - Logical Groups - DDL

```
CREATE TABLE Product (
    ID INT PRIMARY KEY,
    NAME VARCHAR(20) NOT NULL,
    PRICE DECIMAL DEFAULT 0.0
);
```

PRIMARY_KEY: uniquely identifies the sample (row)

FOREIGN_KEY: this row is connected to another table

NOT NULL means the value must NOT be empty (null)

DEFAULT gives a default value if nothing is put in during insertion

```
cursor.execute("""
        CREATE TABLE IF NOT EXISTS staff (
           staffid INTEGER PRIMARY KEY,
           first_name TEXT,
           last_name TEXT,
           trackid INTEGER,
           age INTEGER
        );
""")
```

```
conn.commit()
```

Given the previous DDL statement, we can simply include the CREATE TABLE clause in a triple-quoted string within the **execute** method, but note that we did not actually make any changes to our database just yet!

```
cursor.execute("""
    CREATE TABLE IF NOT EXISTS staff (
      staffid INTEGER PRIMARY KEY,
      first_name TEXT,
      last_name TEXT,
      trackid INTEGER,
      age INTEGER
    );
""")
```

```
conn.commit()
```

Just like with git, we must **commit** any DDL query to the connection if we would like these changes to be reflected! Without a commit the database is left **unmodified**.

SQLite Lab

SQL

Complete the module **lab_part1.py** and **lab_part2.py** modules for the remainder of the lab time

Wednesday

SQL Leetcode Review

- *Another Leetcode day! (pray that it doesn't go down)*
- *We'll hold off on Machine Learning until Phase 2*