LECTURE 21

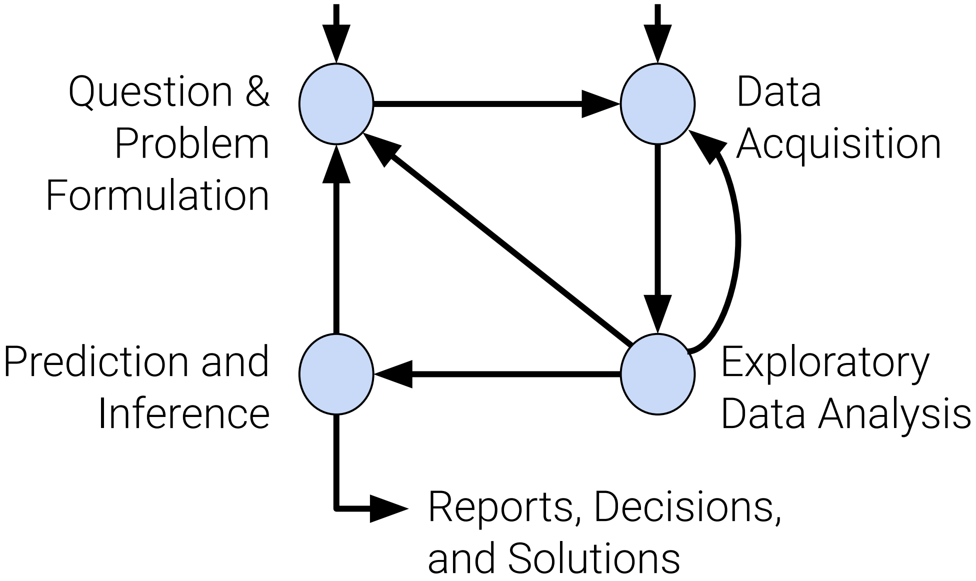
SQL I

SQL and Databases: An alternative to Pandas and CSV files.

Data 100/Data 200, Spring 2023 @ UC Berkeley

Narges Norouzi and Lisa Yan

The Data Science Lifecycle

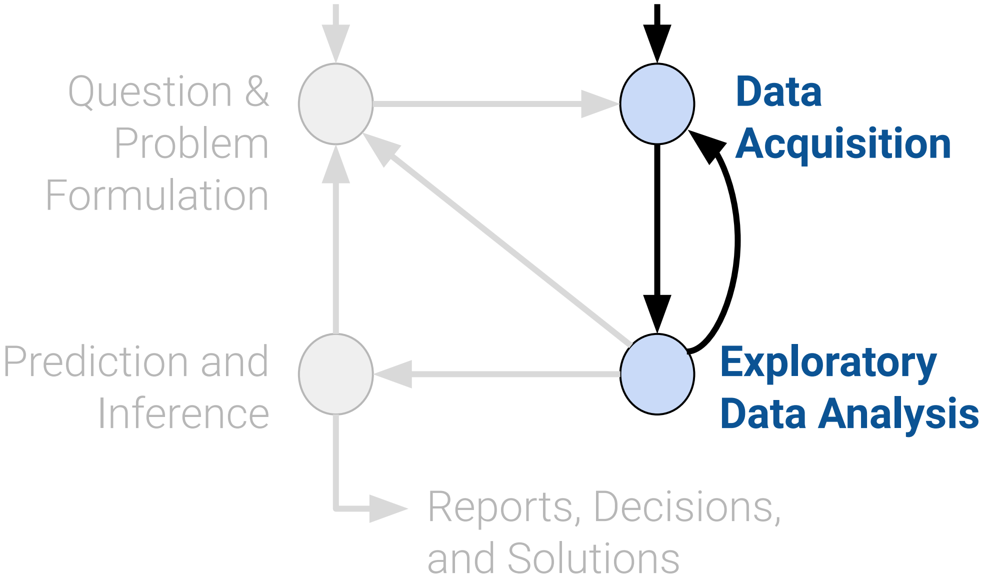


Over the last 10 weeks, we went through the entire data science lifecycle.

In the next four weeks we are going to do it again.

- This time, with a different set of tools, ideas, and abstractions.

SQL and Databases: An Alternative to Pandas and CSV Files



(today)

SQL and Databases I → SQL and Databases II

Today and Part of Thursday

slido

How much experience do you have with SQL?

ⓘ Start presenting to display the poll results on this slide.

Why Databases?

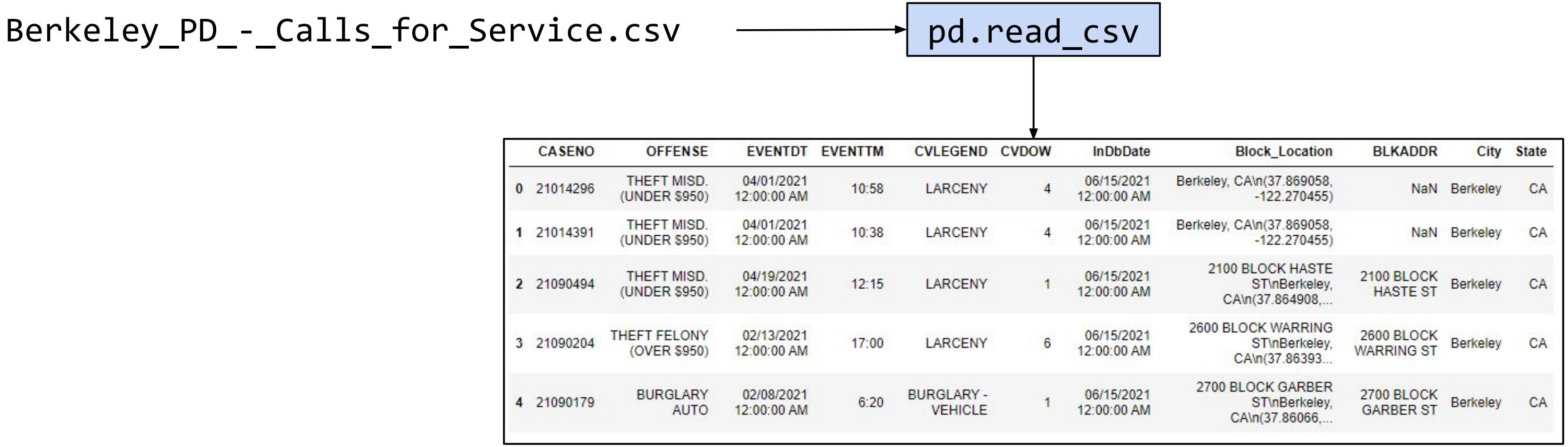
Lecture 21, Data 100 Spring 2023

- **Why Databases**
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- Filtering Groups with HAVING
- DISTINCT
- Python SQL

Weeks 1 - 10: CSV Files and Pandas

1799583

So far in Data 100, we’ve worked with data stored in CSV files.



`Berkeley_PD_-_Calls_for_Service.csv`

`pd.read_csv`

	CASENO	OFFENSE	EVENTDT	EVENTTM	CVLEGEND	CVDOW	InDbDate	Block_Location	BLKADDR	City	State
0	21014296	THEFT MISD. (UNDER $950)	04/01/2021 12:00:00 AM	10:58	LARCENY	4	06/15/2021 12:00:00 AM	Berkeley, CA\n(37.869058, -122.270455)	NaN	Berkeley	CA
1	21014391	THEFT MISD. (UNDER $950)	04/01/2021 12:00:00 AM	10:38	LARCENY	4	06/15/2021 12:00:00 AM	Berkeley, CA\n(37.869058, -122.270455)	NaN	Berkeley	CA
2	21090494	THEFT MISD. (UNDER $950)	04/19/2021 12:00:00 AM	12:15	LARCENY	1	06/15/2021 12:00:00 AM	2100 BLOCK HASTE ST\nBerkeley, CA\n(37.864908,...	2100 BLOCK HASTE ST	Berkeley	CA
3	21090204	THEFT FELONY (OVER $950)	02/13/2021 12:00:00 AM	17:00	LARCENY	6	06/15/2021 12:00:00 AM	2600 BLOCK WARRING ST\nBerkeley, CA\n(37.86393...	2600 BLOCK WARRING ST	Berkeley	CA
4	21090179	BURGLARY AUTO	02/08/2021 12:00:00 AM	6:20	BURGLARY - VEHICLE	1	06/15/2021 12:00:00 AM	2700 BLOCK GARBER ST\nBerkeley, CA\n(37.86066,...	2700 BLOCK GARBER ST	Berkeley	CA

Perfectly reasonable workflow for small data that we’re not actively sharing with others.

Brief Databases Overview

A **database** is an organized collection of data.

A **database management system (DBMS)** is a software system that **stores**, **manages**, and **facilitates access** to one or more databases.

Why use DBMSes?

- Our data might not be stored in a simple-to-read format such as a CSV (comma-separated values) file.
- Think of a CSV like an Excel sheet or a sheet in Google sheets.
- In Data 8, most of the data were given to you in CSV files, but that will not always be the case in the real world.

If our data are stored in a DBMS, we must use languages such as Structured Query Language (SQL) to query for our data.

Advantages of DBMS over CSV (or similar)

1799583

Data Storage:

- **Reliable storage** to survive system crashes and disk failures.
- Optimize to **compute on data that does not fit in memory**.
- Special data structures to **improve performance** (see CS (W)186).

Data Management:

- Configure how data is **logically organized** and **who has access**.
- Can enforce guarantees on the data (e.g. non-negative person weight or age).
  - Can be used to **prevent data anomalies**.
  - Ensures **safe concurrent operations** on data (multiple users reading and writing simultaneously, e.g. ATM transactions).

SQL Overview

Lecture 21, Data 100 Spring 2023

- Why Databases
- **Warmup: SQL Example**
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- DISTINCT
- Python SQL

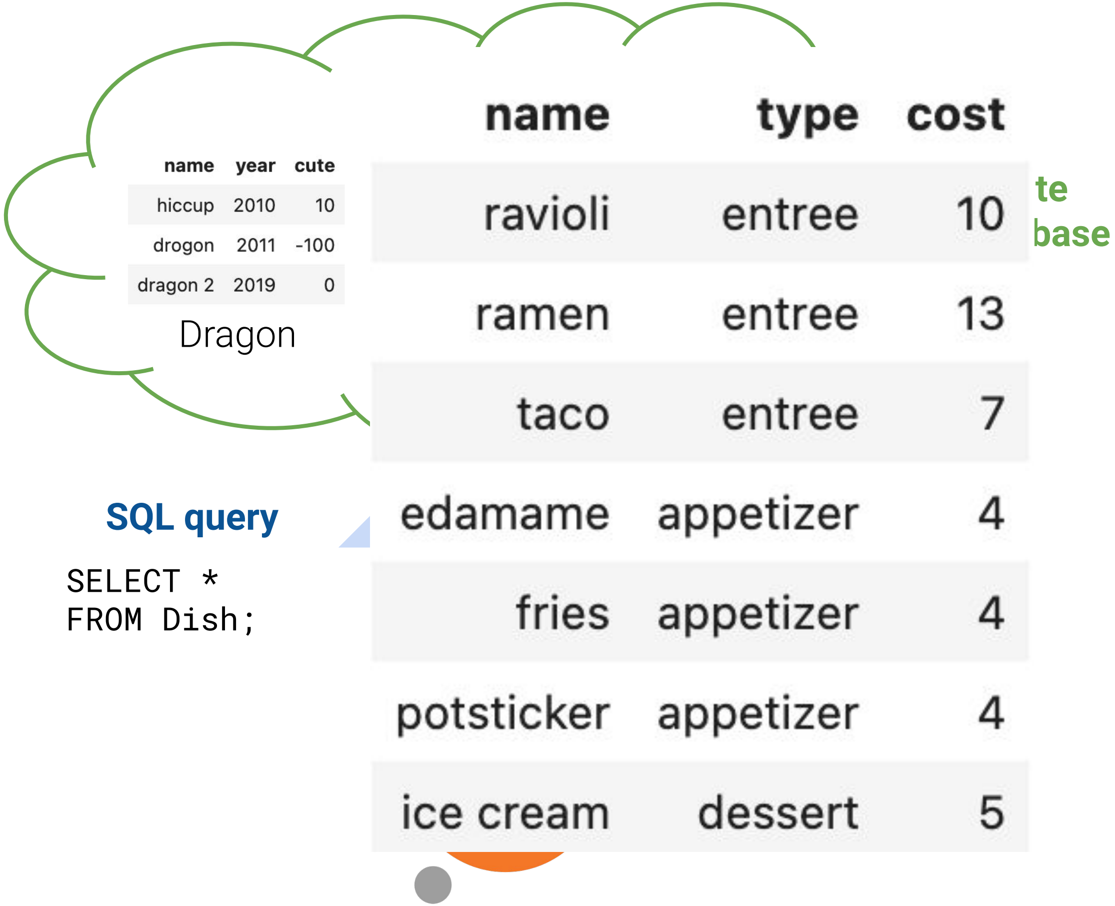
SQL

Today we'll be using a programming language called "Structured Query Language" or SQL.

- SQL is its own programming language, totally distinct from Python.
- SQL is a special purpose programming language used specifically for communicating with databases.
- We will program in SQL using Jupyter notebooks.

Let's see a quick demo of how we can use SQL to connect to a database and view a SQL Table.

Quick SQL Overview



name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

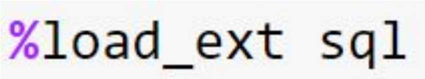
SQL query

```
SELECT *
FROM Dish;
```

Demo

Step 1: Load the SQL Module

Our first step is to load the SQL module. We do so using the `ipython` **cell magic** command:

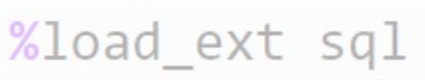


```
%load_ext sql
```

Step 2: Connect to a Database

1799583

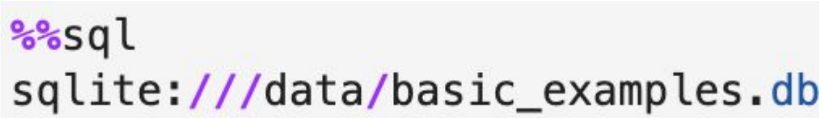
Our first step is to load the SQL module. We do so using the `ipython` **cell magic** command:



```
%load_ext sql
```

The second step is to connect to a database.

We use the `%%sql` header to tell Jupyter that this cell represents SQL code rather than Python code.



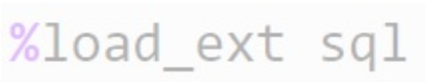
```
%%sql
sqlite:///data/basic_examples.db
```

```
Connected: @data/18_basic_examples.db
```

(A note about SQLite)

1799583

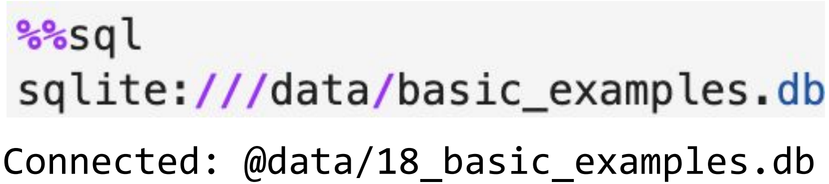
Our first step is to load the SQL module. We do so using the `ipython` **cell magic** command:



```
%load_ext sql
```

The second step is to connect to a database.

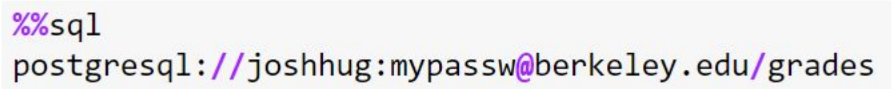
We use the `%%sql` header to tell Jupyter that this cell represents SQL code rather than Python code.



```
%%sql
sqlite:///data/basic_examples.db
```

```
Connected: @data/18_basic_examples.db
```

In Data 100, our database is stored in a local file. In real world practice, you'd probably connect to a remote server.



```
%%sql
postgresql://joshhug:mypassw@berkeley.edu/grades
```

There are various extensions to SQL.

We are learning the SQL commands and syntax supported by the SQLite library.

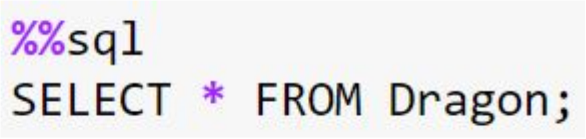
If you're curious: `SQLite` is a library that provides a relational DBMS (RDBMS). It is lightweight and offers file-based databases.

3. Run SQL Statements

1799583

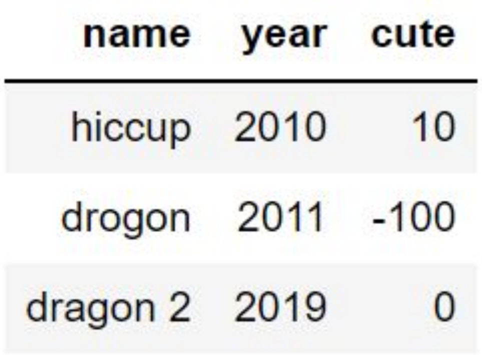
Now that we're connected, let's make some queries!

For example, we might show every row in the `Dragon` table.



```
%%sql
SELECT * FROM Dragon;
```

returns



name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Thanks to the pandas magic, the resulting return data is displayed in a format almost identical to our Pandas tables (without an index).

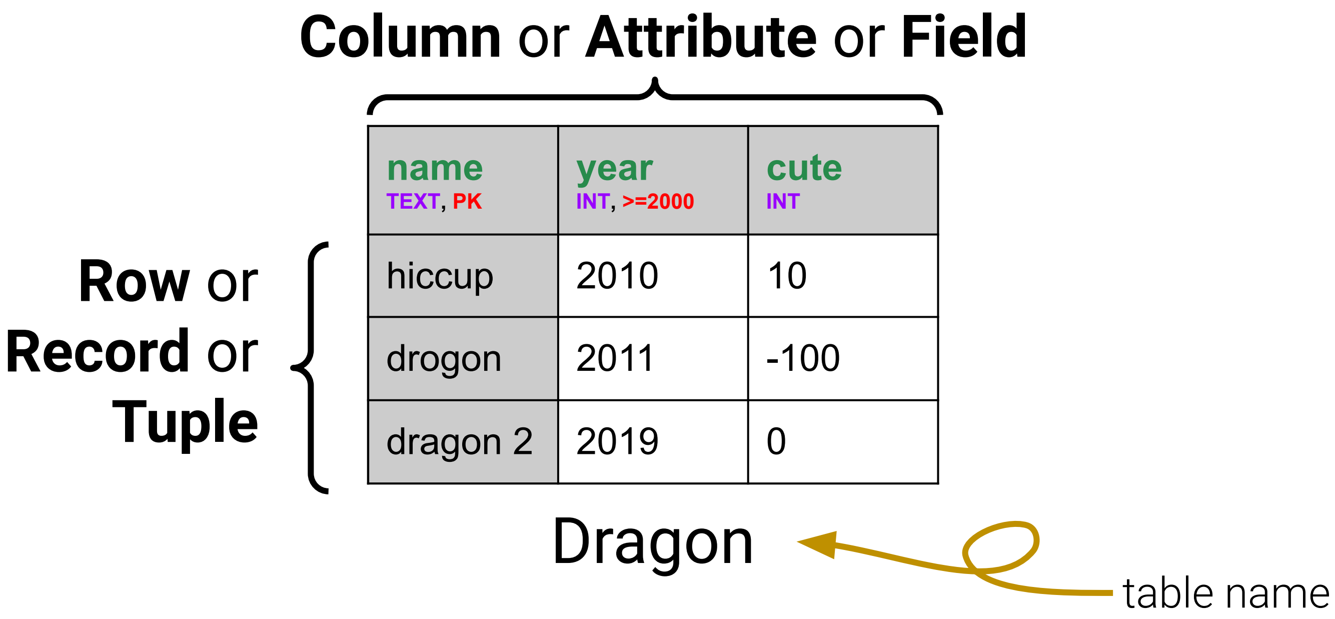
SQL statements are terminated with semicolons. A **SQL query** is a SQL statement that returns data.

SQL Terminology: Schema and Primary Keys

Lecture 21, Data 100 Spring 2023

- Why Databases
- Warmup: SQL Example
- **SQL Tables**
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- Filtering Groups with HAVING
- DISTINCT
- Python SQL

SQL Terminology



Column or **Attribute** or **Field**

Row or **Record** or **Tuple**

name TEXT, PK	year INT, >=2000	cute INT
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

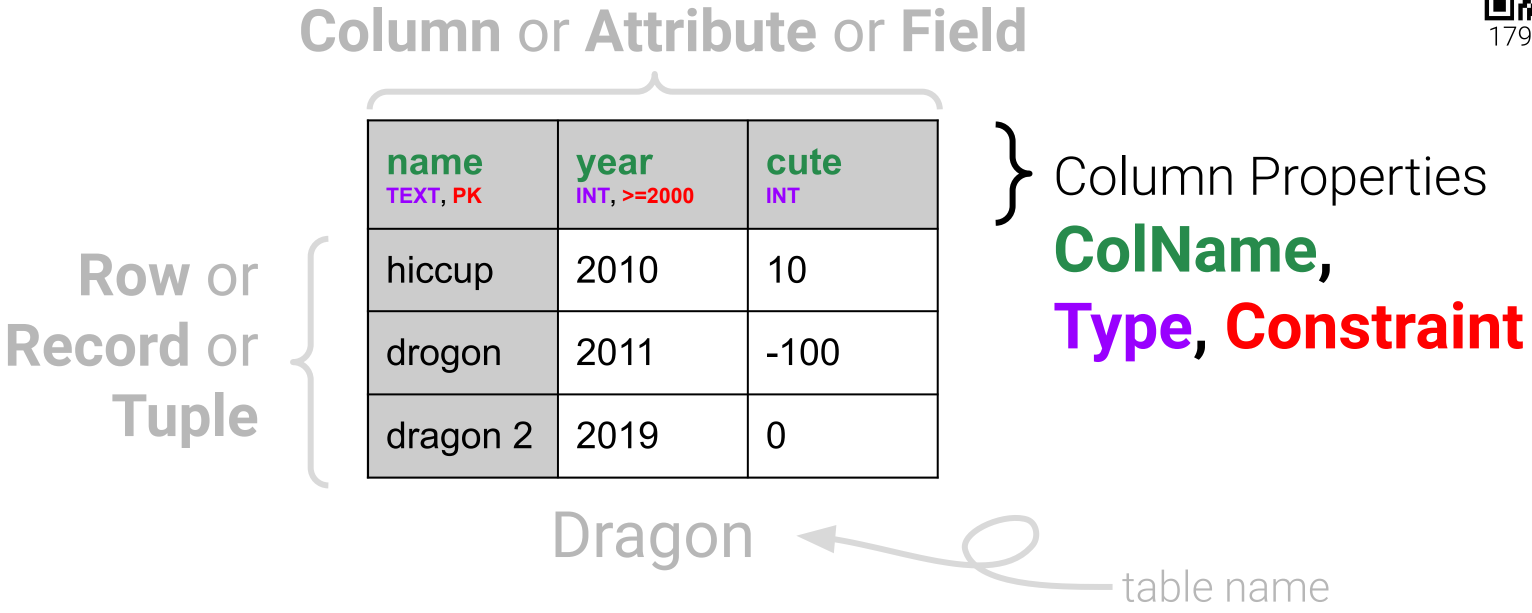
table name

SQL **tables** are also called **relations**.

SQL Style: Use *singular, CamelCase* names for SQL tables! For more, see this post.

SQL Terminology

1799583



Column or Attribute or Field

Row or Record or Tuple

name TEXT, PK	year INT, >=2000	cute INT
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Column Properties

ColName, Type, Constraint

Dragon

table name

SQL **tables** are also called **relations**.

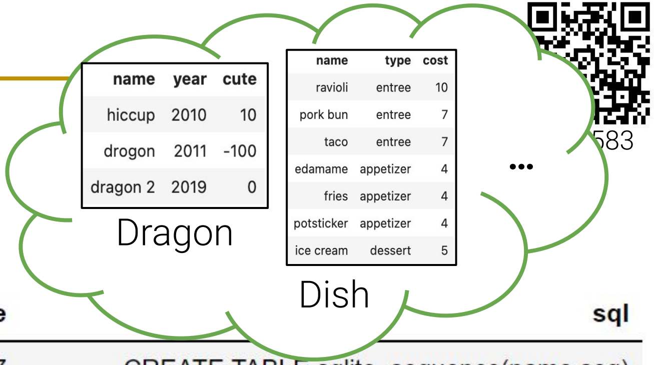
SQL Style: Use *singular, CamelCase* names for SQL tables! For more, see this post.

Every column in a SQL table has three properties: **ColName**, **Type**, and zero or more **Constraints**. (Contrast with Pandas: Series have names and types, but no constraints.)

Table Schemas

583

There are multiple tables in a database:



name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

name	type	cost
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

This list of tables was generated with this SQL query:

```
SELECT *
FROM sqlite_master
WHERE type='table';
```

(Many of the details here are beyond the scope of our class.)

type	name	tbl_name	rootpage	sql
table	sqlite_sequence	sqlite_sequence	7	CREATE TABLE sqlite_sequence(name,seq)
table	Dragon	Dragon	2	CREATE TABLE Dragon (name TEXT PRIMARY KEY, year INTEGER CHECK (year >= 2000), cute INTEGER)
table	Dish	Dish	4	CREATE TABLE Dish (name TEXT PRIMARY KEY, type TEXT, cost INTEGER CHECK (cost >= 0))
table	Scene	Scene	6	CREATE TABLE Scene (id INTEGER PRIMARY KEY AUTOINCREMENT, biome TEXT NOT NULL, city TEXT NOT NULL, visitors INTEGER CHECK (visitors >= 0), created_at DATETIME DEFAULT (DATETIME('now')))

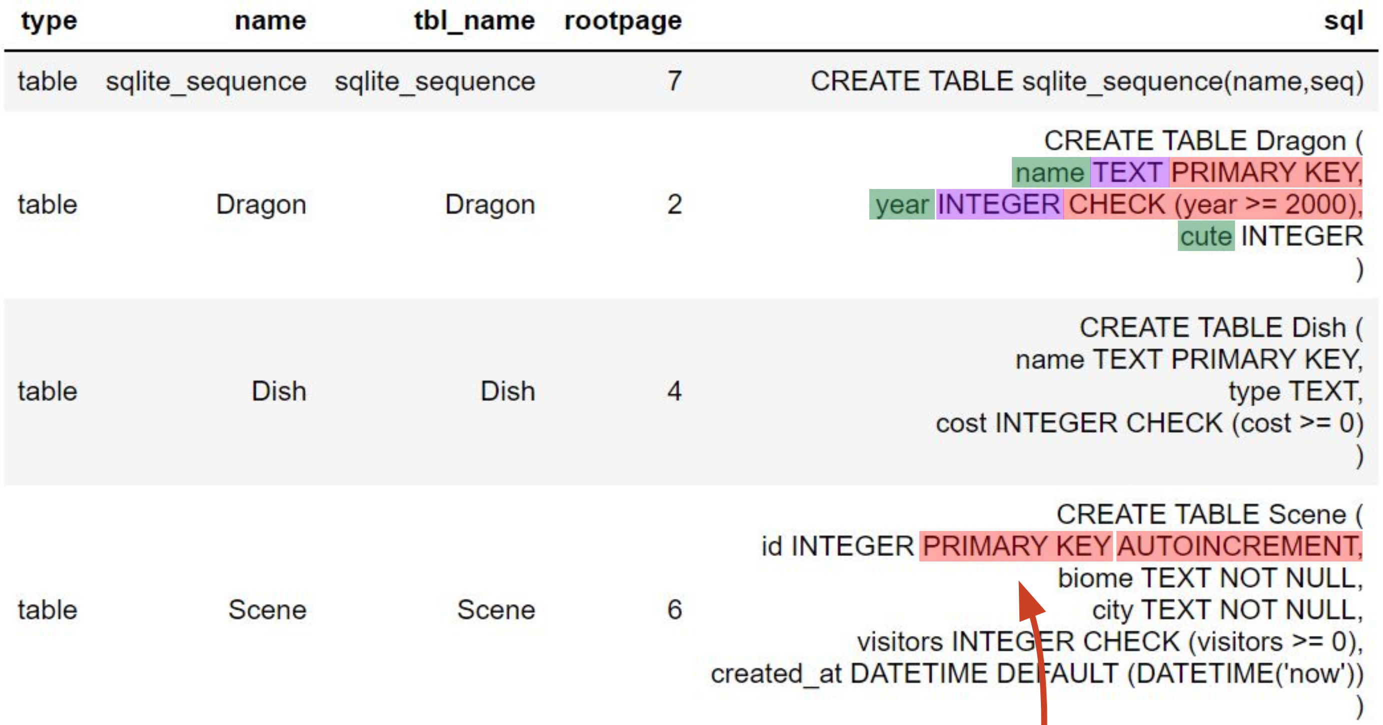
The "**sql**" column gives the command used to create each table, and by doing so shows us each **table schema**.

Table Schemas

1799583

The table schema specifies each column schema.

Every column in a SQL table has three properties: **ColName, Type**, and zero or more **Constraints**.



type	name	tbl_name	rootpage	sql
table	sqlite_sequence	sqlite_sequence	7	CREATE TABLE sqlite_sequence(name,seq)
table	Dragon	Dragon	2	CREATE TABLE Dragon (name TEXT PRIMARY KEY, year INTEGER CHECK (year >= 2000), cute INTEGER)
table	Dish	Dish	4	CREATE TABLE Dish (name TEXT PRIMARY KEY, type TEXT, cost INTEGER CHECK (cost >= 0))
table	Scene	Scene	6	CREATE TABLE Scene (id INTEGER PRIMARY KEY AUTOINCREMENT, biome TEXT NOT NULL, city TEXT NOT NULL, visitors INTEGER CHECK (visitors >= 0), created_at DATETIME DEFAULT (DATETIME('now')))

2 **constraints**

Example **Types**

1799583

Some examples of SQL **types**:

- `INT`: Integers.
- `REAL`: Real numbers.
- `TEXT`: Strings of text.
- `BLOB`: Arbitrary data, e.g. songs, video files, etc.
- `DATETIME`: A date and time.

Note: Different implementations of SQL support different types.

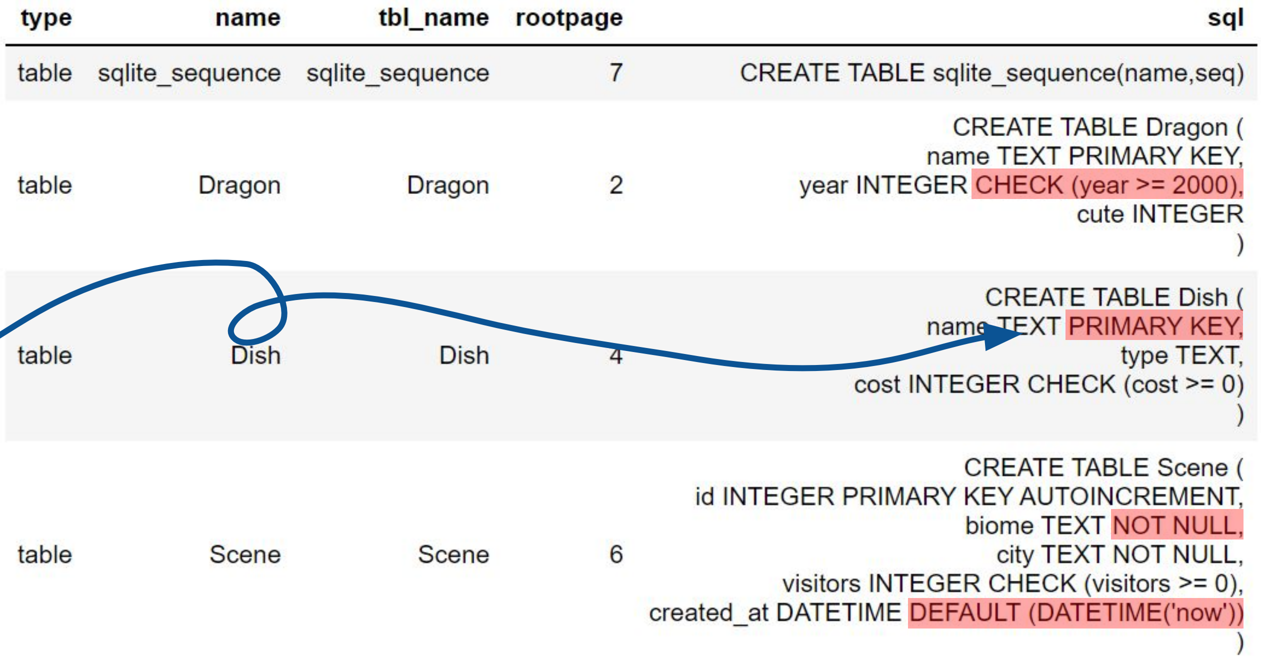
- SQLite: https://www.sqlite.org/datatype3.html
- MySQL: https://dev.mysql.com/doc/refman/8.0/en/data-types.html

Example Constraints

1799583

Some examples of **constraints**:

- `CHECK`: Data cannot be inserted which violates the given check constraint.
- `PRIMARY KEY`: Specifies that this key is used to uniquely identify rows in the table.
- `NOT NULL`: Null data cannot be inserted for this column.
- `DEFAULT`: Provides a value to use if user does not specify on insertion.



type	name	tbl_name	rootpage	sql
table	sqlite_sequence	sqlite_sequence	7	CREATE TABLE sqlite_sequence(name,seq)
table	Dragon	Dragon	2	CREATE TABLE Dragon (name TEXT PRIMARY KEY, year INTEGER CHECK (year >= 2000), cute INTEGER)
table	Dish	Dish	4	CREATE TABLE Dish (name TEXT PRIMARY KEY, type TEXT, cost INTEGER CHECK (cost >= 0))
table	Scene	Scene	6	CREATE TABLE Scene (id INTEGER PRIMARY KEY AUTOINCREMENT, biome TEXT NOT NULL, city TEXT NOT NULL, visitors INTEGER CHECK (visitors >= 0), created_at DATETIME DEFAULT (DATETIME('now')))

What is this primary key constraint?

Primary Keys

A **primary key** is used to uniquely identify each record in the table.

- In the Dragon table, the "**name**" of each Dragon is the primary key.
- In other words, no two dragons can have the same name!
- Primary key is used **under the hood** for all sorts of optimizations.

name **TEXT**, **PK**	**year** **INT**, **>=2000**	**cute** **INT**
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Why specify primary keys? More next time when we discuss JOINs...

Primary Keys Can Span Columns

A **primary key** is used to uniquely identify each record in the table.

- In the Dragon table, the "**name**" of each Dragon is the primary key.
- In other words, no two dragons can have the same name!
- Primary key is used **under the hood** for all sorts of optimizations.

name TEXT, PK	**year** INT, >=2000	**cute** INT
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

The primary key is a **constraint**. A table can have **multiple columns** marked as primary key:

- **No duplicate tuples** allowed across the primary keys.
- Ok to have two "Serral" or two "Starcraft 2" rows.
- May only have one row with both "Serral" and "Starcraft 2".

progamer TEXT, PK	**game** TEXT, PK	**earnings** REAL, >=0
Flash	Starcraft	580,305.13
Flash	Starcraft 2	90,152.64
Jaedong	Starcraft	418,456.82
Serral	Starcraft 2	1,143,488.55
Jaedong	Starcraft 2	224,833.53

Two video games that have pro esports players.

Basic SQL Queries

Lecture 21, Data 100 Spring 2023

- Why Databases
- Warmup: SQL Example
- SQL Tables
- **Basic SQL Queries**
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- Filtering Groups with HAVING
- DISTINCT
- Python SQL

Summary So Far

Query Syntax So Far

```
SELECT <column list>
FROM <table>
```

```
;
```

Marks the end of a SQL statement.

Goal of this section

New keywords

```
SELECT <column list>
FROM <table>
[WHERE <predicate>]
[ORDER BY <column list>]
[LIMIT <number of rows>]
[OFFSET <number of rows>];
```

By the end of this section, you will learn these new keywords!

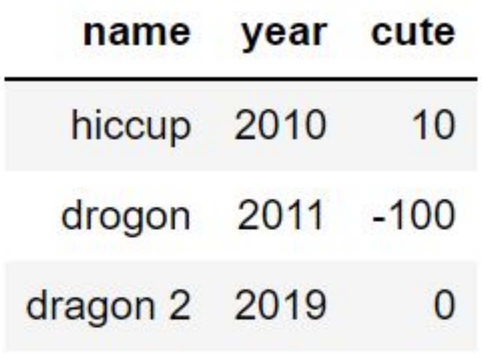
But first, more SELECT

1799583

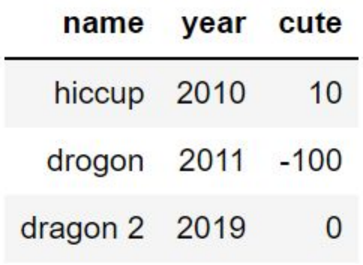
Recall our simplest query, which returns the full relation:

```
SELECT *
FROM Dragon;
```

table name



name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0



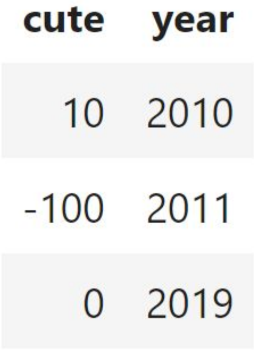
name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

We can also `SELECT` only a **subset of the columns**:

column expression list

```
SELECT cute, year
FROM Dragon;
```



cute	year
10	2010
-100	2011
0	2019

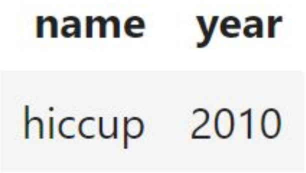
WHERE: Select a rows based on conditions

1799583

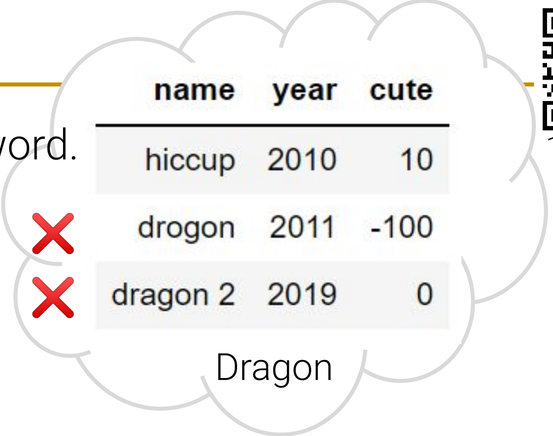
To select only some rows of a table, we can use the WHERE keyword.

```
SELECT name, year
FROM Dragon
WHERE cute > 0;
```

condition



name	year
hiccup	2010



	name	year	cute
	hiccup	2010	10
❌	drogon	2011	-100
❌	dragon 2	2019	0

Dragon

Mnemonic device for later:

“The Row WHERE” Dragon

WHERE: Select a rows based on conditions

1799583

To select only some rows of a table, we can use the WHERE keyword.

```
SELECT name, year
FROM Dragon
WHERE cute > 0;
```

condition

name	year
hiccup	2010

	name	year	cute
	hiccup	2010	10
❌	drogon	2011	-100
	dragon 2	2019	0

Dragon

Mnemonic device for later:

“The Row WHERE” Dragon

The OR, AND, and NOT let us form more complex conditions.

```
SELECT name, year
FROM Dragon
WHERE cute > 0 OR year > 2013;
```

condition

name	year
hiccup	2010
dragon 2	2019

(fixed post-lecture. Only name, year columns returned)

ORDER BY: Sort rows

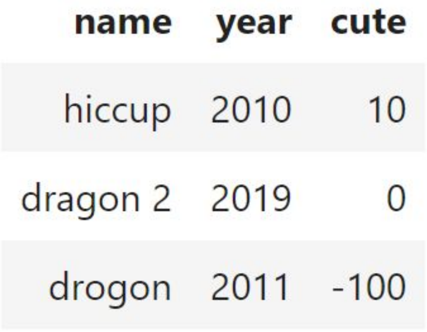
1799583

Self-explanatory

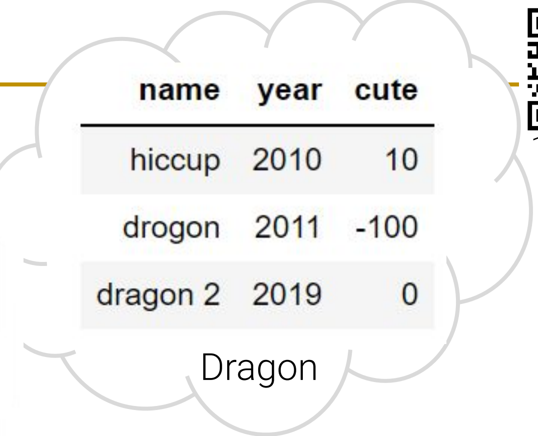
```
SELECT *
FROM Dragon
ORDER BY cute DESC;
```

column

(or **ASC**)



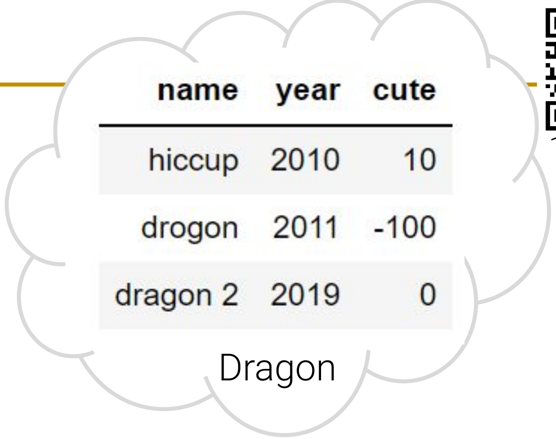
name	year	cute
hiccup	2010	10
dragon 2	2019	0
drogon	2011	-100



name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

OFFSET vs. LIMIT?



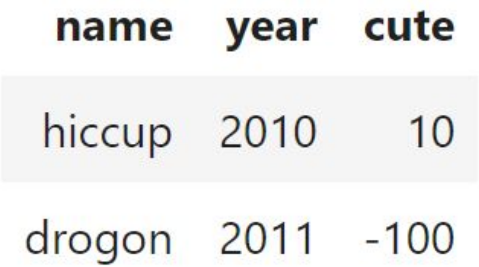
name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

1799583

1.
```
SELECT *
FROM Dragon
LIMIT 2;
```

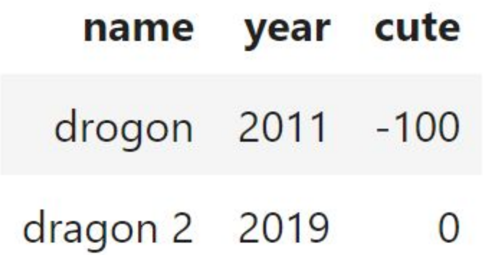
A.



name	year	cute
hiccup	2010	10
drogon	2011	-100

2.
```
SELECT *
FROM Dragon
LIMIT 2
OFFSET 1;
```

B.



name	year	cute
drogon	2011	-100
dragon 2	2019	0

Matching: Which query matches each relation?

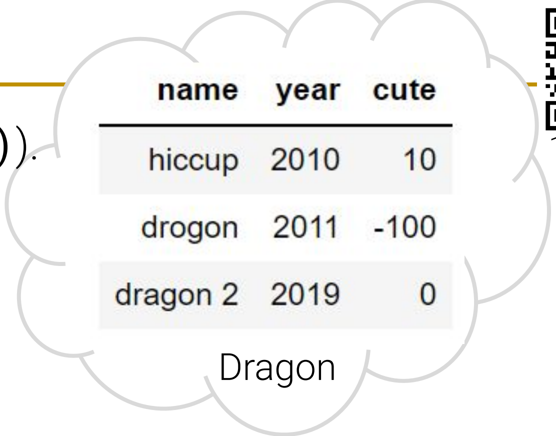
(no Slido) What do you think the **LIMIT** and **OFFSET** keywords do?

slido

Matching: Which query matches each relation?

ⓘ Start presenting to display the poll results on this slide.

OFFSET and LIMIT



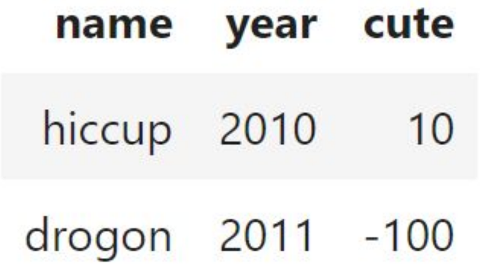
name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

1799583

The LIMIT keyword lets you retrieve N rows (like pandas **head()**).

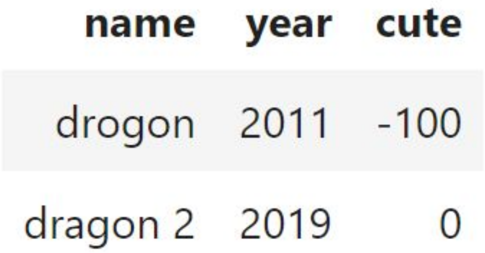
```
SELECT *
FROM Dragon
LIMIT 2;
```



name	year	cute
hiccup	2010	10
drogon	2011	-100

The OFFSET keyword lets you tell SQL to see later rows when limiting.

```
SELECT *
FROM Dragon
LIMIT 2
OFFSET 1;
```



name	year	cute
drogon	2011	-100
dragon 2	2019	0

⚠️ Unless you use ORDER BY, there is **no guaranteed order** of rows in the relation!

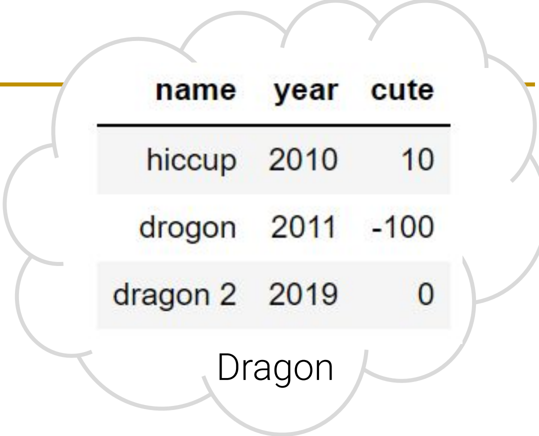
Summary So Far

New keywords

```
SELECT <column list>
FROM <table>
[WHERE <predicate>]
[ORDER BY <column list>]
[LIMIT <number of rows>]
[OFFSET <number of rows>];
```

- By convention, use **all caps** for keywords in SQL statements.

The AS Keyword Aliases Columns



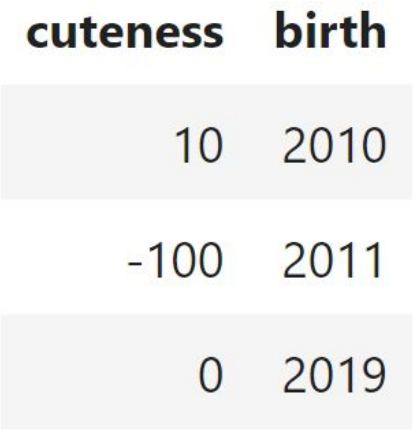
name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0

Dragon

1799583

The AS keyword lets us rename columns during the selection process:

```
SELECT cute AS cuteness,
       year AS birth
FROM Dragon;
```



cuteness	birth
10	2010
-100	2011
0	2019

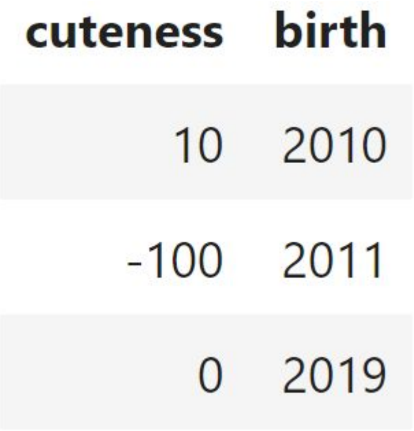
The AS keyword **aliases** column names.

SQL Style: Newline Separators

The following two queries both retrieve the same relation:

```
SELECT cute AS cuteness,
       year AS birth
FROM Dragon;
```

(more readable)



cuteness	birth
10	2010
-100	2011
0	2019

```
SELECT cute AS
cuteness, year AS
birth FROM Dragon;
```

Use newlines and whitespace wisely in your SQL queries.
It will simplify your debugging process!

Summary So Far

New SELECT expressions

```
SELECT <column expression list>
FROM <table>
[WHERE <predicate>]
[ORDER BY <column list>]
[LIMIT <number of rows>]
[OFFSET <number of rows>];
```

- By convention, use **all caps** for keywords in SQL statements.
- Use **newlines** to make SQL code more readable.
- **AS keyword: rename columns during selection process.**

Basic GROUP BY Operations

Lecture 21, Data 100 Spring 2023

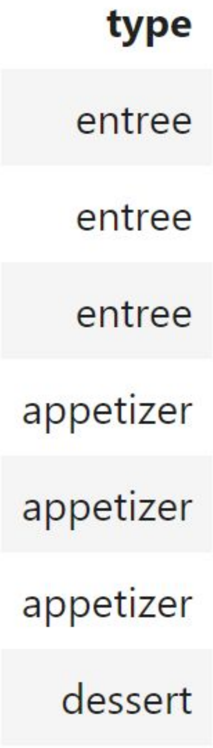
- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- **Basic GROUP BY Operations**
- Trickier GROUP BY Operations
- Filtering Groups with HAVING
- DISTINCT
- Python SQL

The Dish Table

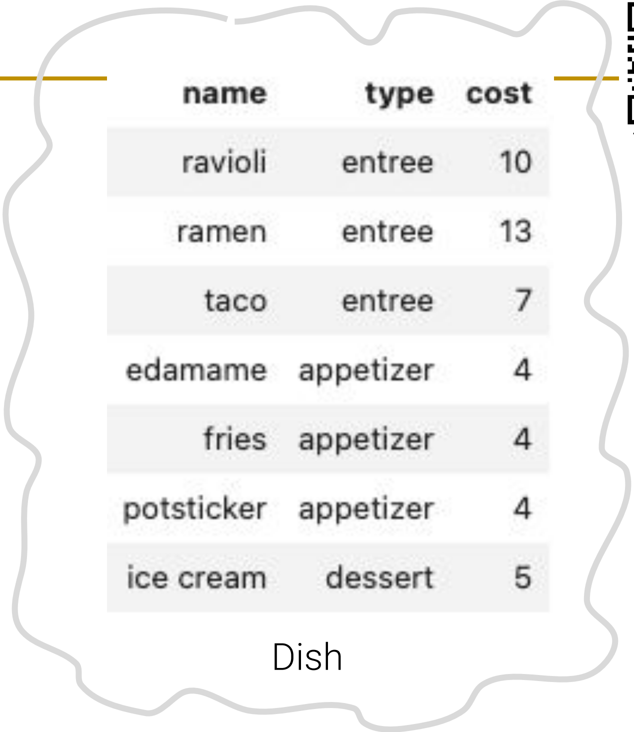
1799583

```
SELECT type
FROM Dish;
```

type is not a SQL keyword.



type
entree
entree
entree
appetizer
appetizer
appetizer
dessert



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

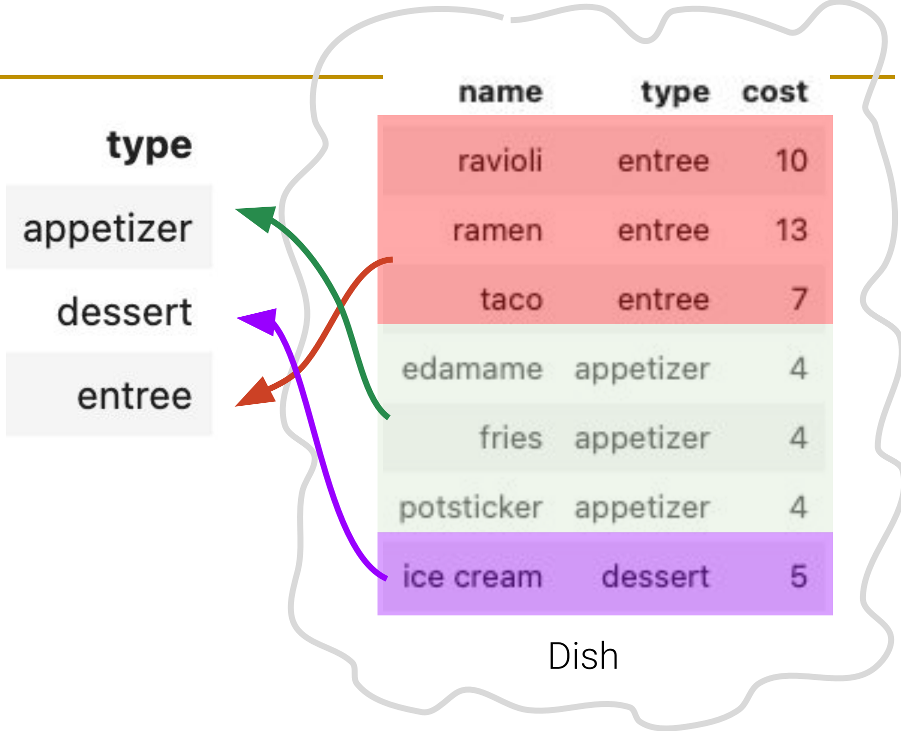
How do we query for a relation that **groups together** dishes of the same type?

GROUP BY

1799583

`GROUP BY` is similar to pandas `groupby()`.

```
SELECT type
FROM Dragon
GROUP BY type;
```



type
appetizer
dessert
entree

name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

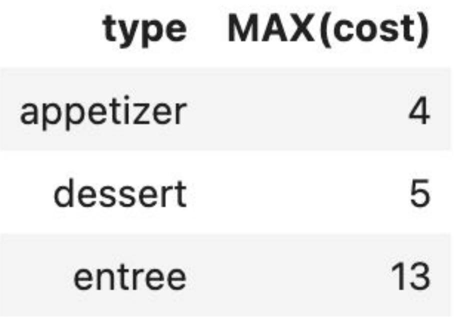
Dish

Aggregate Functions in Column Expression Lists

1799583

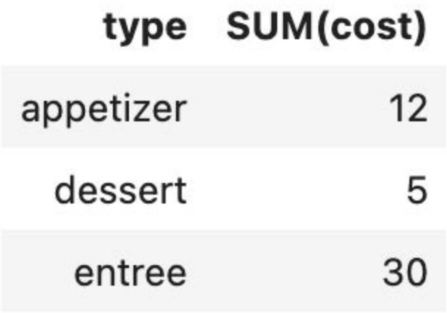
SQL has **aggregate functions**: MAX, SUM, etc.
Similar to pandas `groupby().max()`, etc.

```
SELECT type,
       MAX(cost)
FROM Dish
GROUP BY type;
```

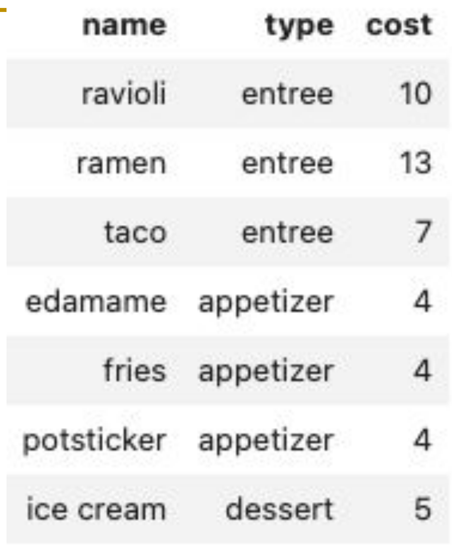


type	MAX(cost)
appetizer	4
dessert	5
entree	13

```
SELECT type,
       SUM(cost)
FROM Dish
GROUP BY type;
```



type	SUM(cost)
appetizer	12
dessert	5
entree	30



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

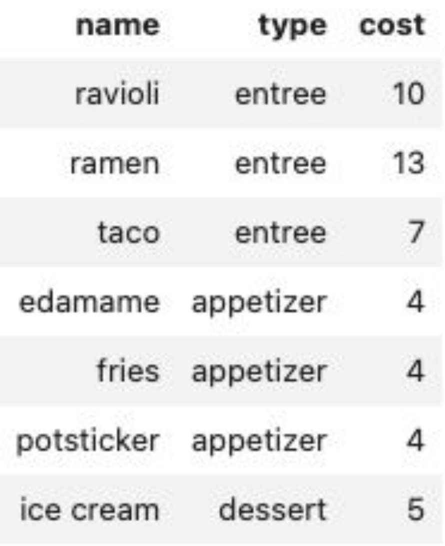
Dish

For more aggregation functions see
https://www.sqlite.org/lang_aggfunc.html

Using Multiple Aggregation Functions

```
SELECT type,
       SUM(cost),
       MIN(cost),
       MAX(name)
FROM Dish
GROUP BY type;
```

What do you think will happen?



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

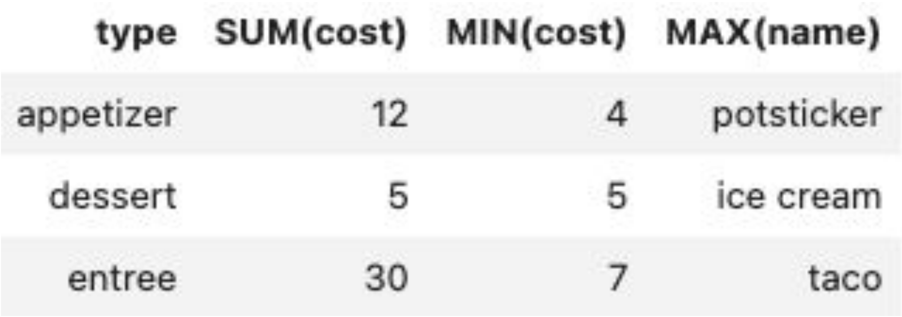
Dish

1799583

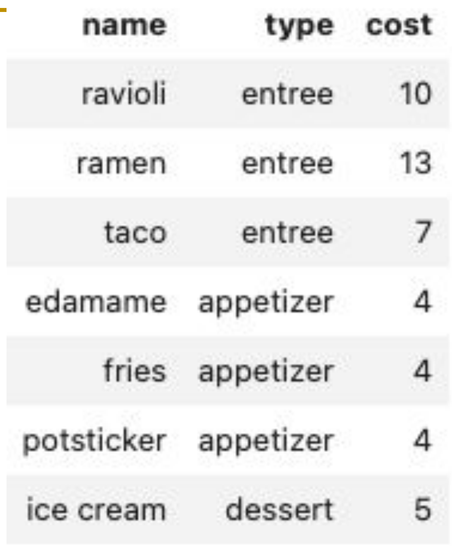
(no slido)

Using Multiple Aggregation Functions

```
SELECT type,
       SUM(cost),
       MIN(cost),
       MAX(name)
FROM Dish
GROUP BY type;
```



type	SUM(cost)	MIN(cost)	MAX(name)
appetizer	12	4	potsticker
dessert	5	5	ice cream
entree	30	7	taco



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

1799583

No simple equivalent in pandas!

New keywords

```
SELECT <column expression list>
FROM <table>
[WHERE <predicate>]
[GROUP BY <column list>]
[ORDER BY <column list>]
[LIMIT <number of rows>]
[OFFSET <number of rows>];
```

Summary So Far

- By convention, use **all caps** for keywords in SQL statements.
- Use **newlines** to make SQL code more readable.
- **AS** keyword: rename columns during selection process.
- **Column Expressions may include aggregation functions (`MAX`, `MIN`, etc.)**

Trickier GROUP BY Operations

Lecture 21, Data 100 Spring 2023

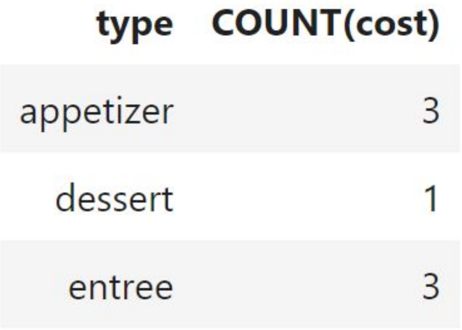
- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- **Trickier GROUP BY Operations**
- Filtering Groups with HAVING
- DISTINCT
- Python SQL

COUNT(colname) vs COUNT(*)

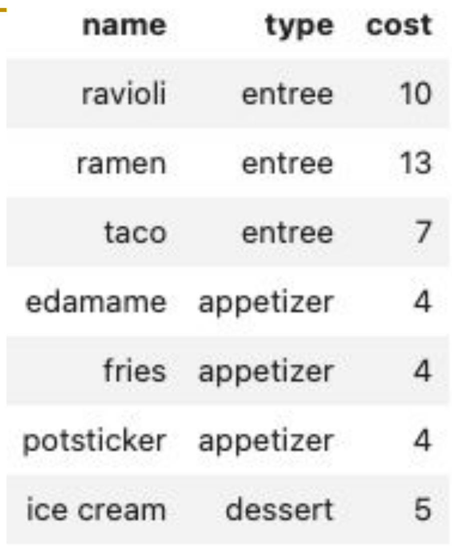
1799583

```
SELECT type, COUNT(cost)
FROM Dish
GROUP BY type;
```

similar to pandas
groupby().count()



type	COUNT(cost)
appetizer	3
dessert	1
entree	3

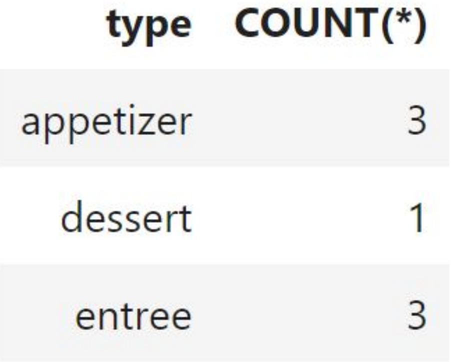


name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

```
SELECT type, COUNT(*)
FROM Dish
GROUP BY type;
```

similar to pandas
groupby().size()



type	COUNT(*)
appetizer	3
dessert	1
entree	3

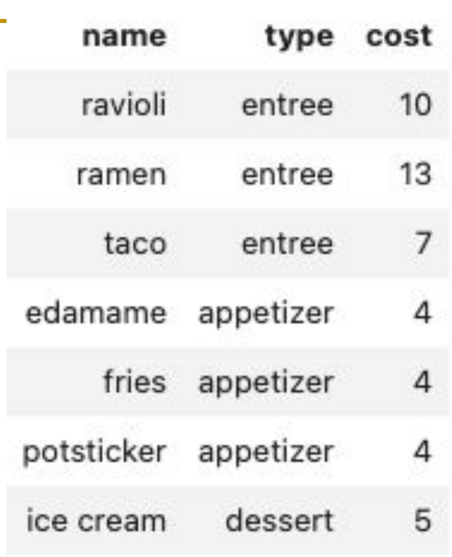
`COUNT(*)` returns the number of rows in each group, including rows with **nulls**.

⚠️ GROUP BY Behavior

1799583

What if we GROUP BY **without** specifying enough aggregation functions in the select expression list?

```
SELECT type, cost
FROM Dish
GROUP BY type;
```



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

Implementation dependent:

- In some variants of SQL, this is allowed.
- In other variants, it is a syntax error.

MS SQL:

> Column Dish.cost' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

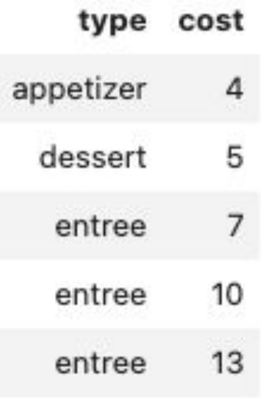
This is considered bad practice. Avoid!!!

GROUP BY to get unique tuples?

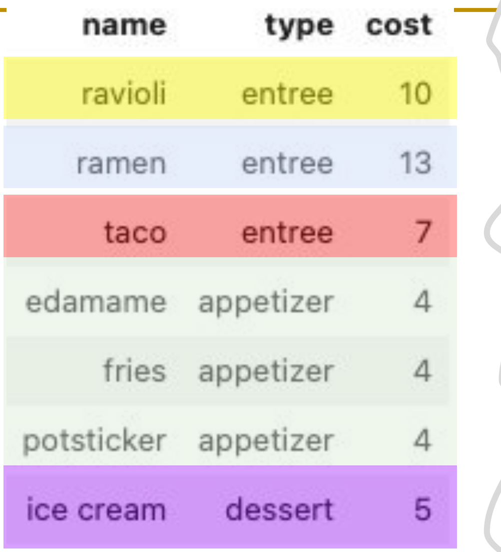
1799583

Like in pandas, you can `GROUP BY` multiple columns.

```
SELECT type, cost
FROM Dish
GROUP BY type, cost;
```



type	cost
appetizer	4
dessert	5
entree	7
entree	10
entree	13



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

This works because **# cols selected = # cols to group on**, but it is a bit unwieldy. We'll learn another approach soon…

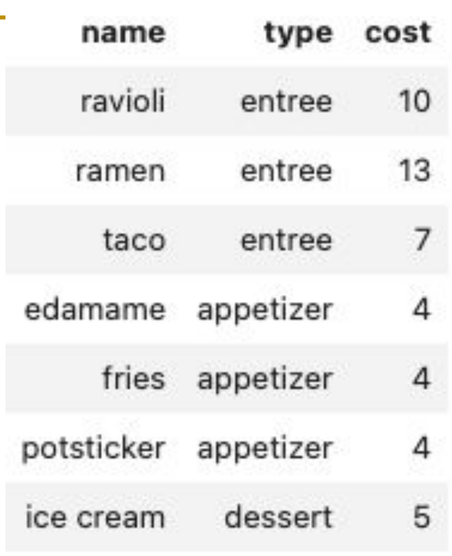
Exercise: GROUP BY Multiple Columns

1799583

A.
```
SELECT type, cost
FROM Dish
GROUP BY type, cost, COUNT(*);
```

B.
```
SELECT type, cost, COUNT(*)
FROM Dish
GROUP BY type, cost;
```

How would we add a third column giving us the number of rows that match each type/cost tuple?



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

slido

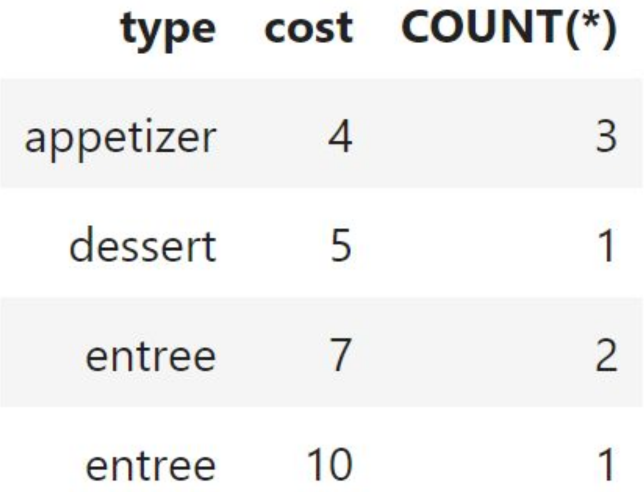
How would we add a third column giving us the number of rows that match each type/cost tuple?

ⓘ Start presenting to display the poll results on this slide.

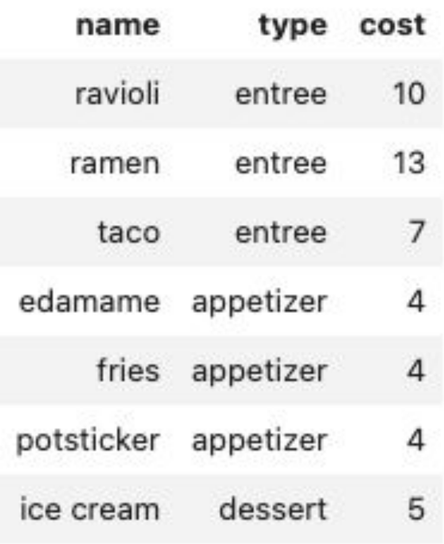
Exercise: GROUP BY Multiple Columns

1799583

B.
```
SELECT type, cost, COUNT(*)
FROM Dish
GROUP BY type, cost;
```



type	cost	COUNT(*)
appetizer	4	3
dessert	5	1
entree	7	2
entree	10	1



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

Filter Groups with HAVING

Lecture 21, Data 100 Spring 2023

- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- **Filtering Groups with HAVING**
- DISTINCT
- Python SQL

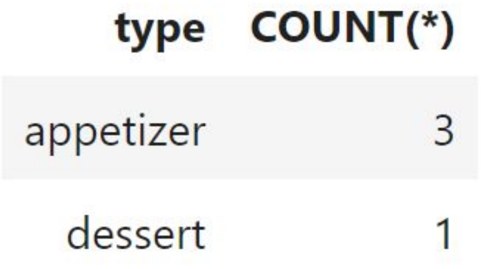
Recall earlier mnemonic device:

“The Row WHERE” Dragon

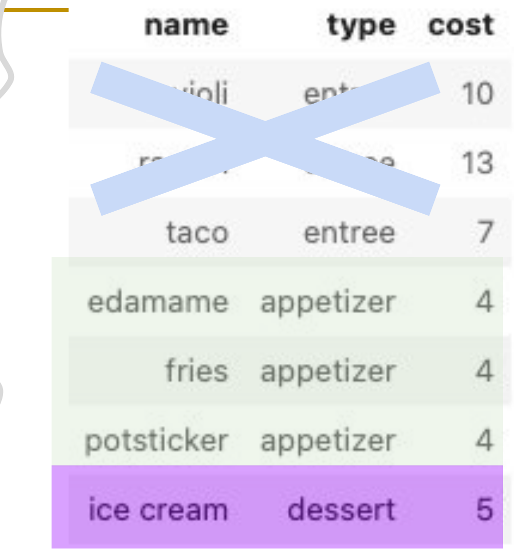
Filtering Groups with HAVING

1799583

```
SELECT type, COUNT(*)
FROM Dish
GROUP BY type
HAVING MAX(cost) < 8;
```



type	COUNT(*)
appetizer	3
dessert	1



name	type	cost
~~ravioli~~	~~entree~~	~~10~~
~~ramen~~	~~entree~~	~~13~~
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

similar to `groupby("type")`
```
                .filter(lambda f: max(f["cost"]) < 8)
```

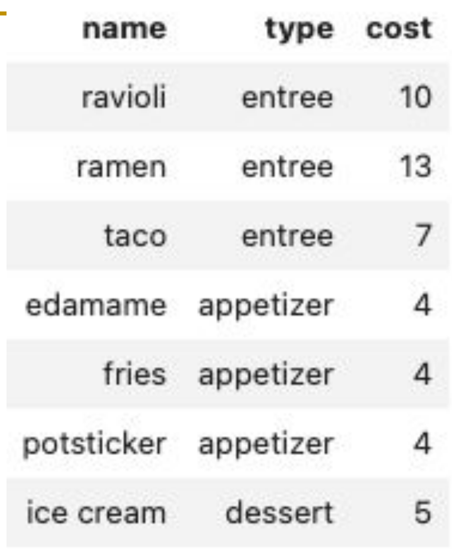
Mnemonic device:

"The Group HAVING" Fish

WHERE vs. HAVING

```
SELECT type, COUNT(*)
FROM Dish
WHERE cost < 8
GROUP BY type;
```

What will happen here?



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

1799583

“The Group HAVING” Fish

“The Row WHERE” Dragon

(no slido)

WHERE vs. HAVING

1799583

```
SELECT type, COUNT(*)
FROM Dish
WHERE cost < 8
GROUP BY type;
```

type	COUNT(*)
appetizer	3
dessert	1
entree	1

	name	type	cost
❌	ravioli	entree	10
❌	ramen	entree	13
	taco	entree	7
	edamame	appetizer	4
	fries	appetizer	4
	potsticker	appetizer	4
	ice cream	dessert	5

Dish

“The Group HAVING” Fish

“The Row WHERE” Dragon

Animation: HAVING vs. WHERE

1799583

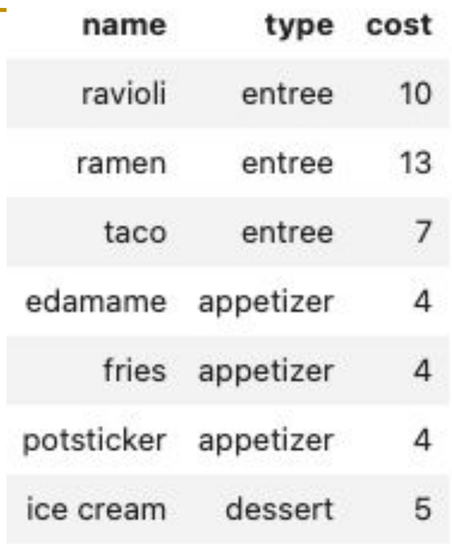
To filter:

- Rows, use **WHERE**.
- Groups, use **HAVING**.

WHERE precedes **HAVING**.

"The Group HAVING" Fish

"The Row WHERE" Dragon



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

```
SELECT ...
WHERE ...
GROUP BY ...
HAVING ...
```

Animation: HAVING vs. WHERE

1799583

To filter:

- Rows, use **WHERE**.
- Groups, use **HAVING**.

WHERE precedes **HAVING**.

“The Group HAVING” Fish

“The Row WHERE” Dragon

	name	type	cost
✗	ravioli	entree	10
	ramen	entree	13
✗	taco	entree	7
	edamame	appetizer	4
	fries	appetizer	4
✗	potsticker	appetizer	4
	ice cream	dessert	5

Dish

```
SELECT ...
WHERE ...
GROUP BY ...
HAVING ...
```

Animation: HAVING vs. WHERE

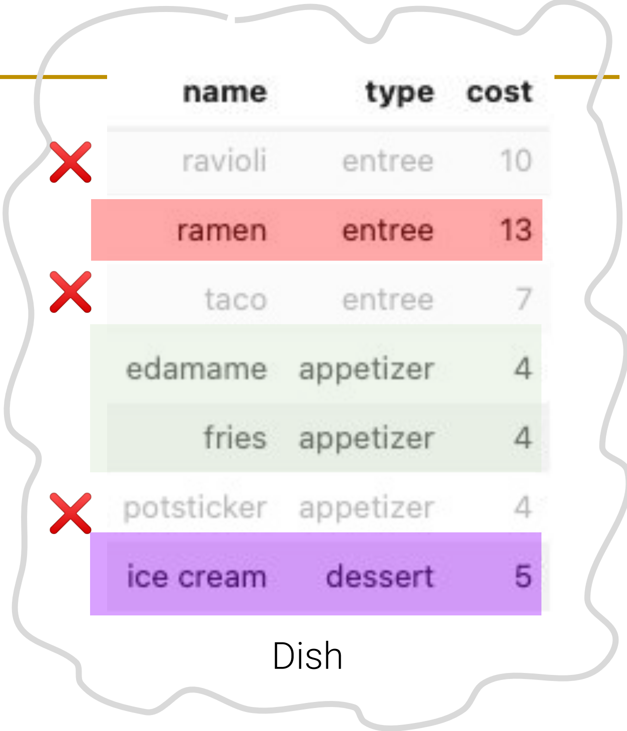
To filter:

- Rows, use **WHERE**.
- Groups, use **HAVING**.

WHERE precedes **HAVING**.

“The Group HAVING” Fish

“The Row WHERE” Dragon



	name	type	cost
❌	ravioli	entree	10
	ramen	entree	13
❌	taco	entree	7
	edamame	appetizer	4
	fries	appetizer	4
❌	potsticker	appetizer	4
	ice cream	dessert	5

Dish

1799583

```
SELECT ...
WHERE ...
GROUP BY ...
HAVING ...
```

Animation: HAVING vs. WHERE

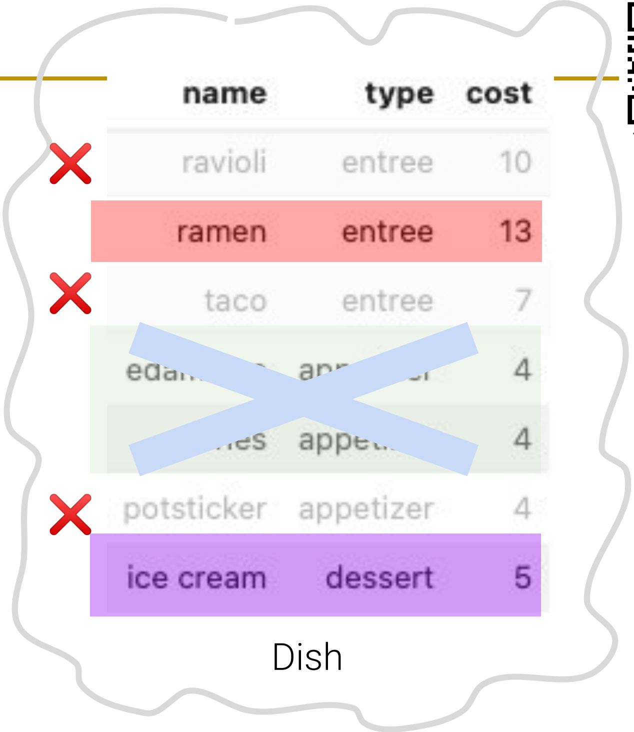
To filter:

- Rows, use **WHERE**.
- Groups, use **HAVING**.

WHERE precedes **HAVING**.

“The Group HAVING” Fish

“The Row WHERE” Dragon



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
eda[illegible]	ap[illegible]	4
[illegible]es	appe[illegible]	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

```
SELECT ...
WHERE ...
GROUP BY ...
HAVING ...
```

Summary So Far

New keywords

```
SELECT <column expression list>
FROM <table>
[WHERE <predicate>]
[GROUP BY <column list>]
[HAVING <predicate>]
[ORDER BY <column list>]
[LIMIT <number of rows>]
[OFFSET <number of rows>];
```

- By convention, use **all caps** for keywords in SQL statements.
- Use **newlines** to make SQL code more readable.
- **AS** keyword: rename columns during selection process.
- **WHERE: rows; HAVING: groups. WHERE precedes HAVING.**

Got to here.
Will cover on
Thursday 4/6

DISTINCT

Lecture 21, Data 100 Spring 2023

- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- Filtering Groups with HAVING
- **DISTINCT**
- Python SQL

Order of Execution

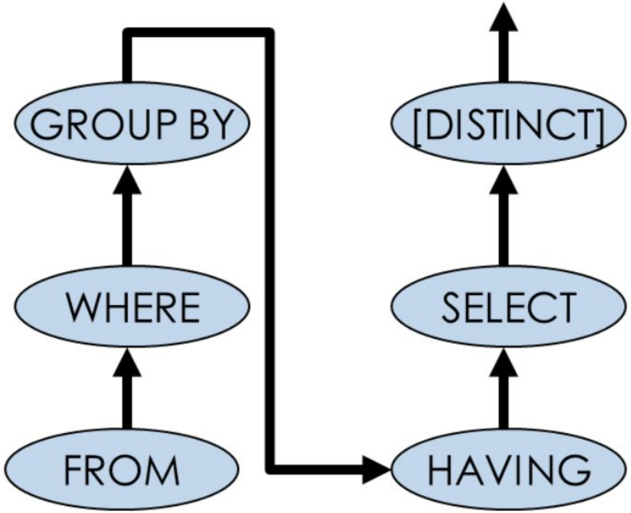
1799583

A query is **not** evaluated according to Python operator precedence.

Generally, the order of execution of clauses within a statement are:

1. `FROM`: Retrieve the Relations.
2. `WHERE`: Filter the rows.
3. `GROUP BY`: Make groups.
4. `HAVING`: Filter the groups.
5. `SELECT`: aggregate into rows, get specific columns.
6. `DISTINCT`: ???

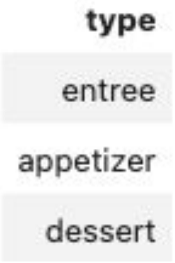
Let’s check it out!



DISTINCT: What does this do?

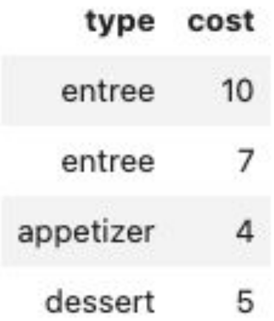
1799583

```
SELECT DISTINCT type
FROM Dish
WHERE cost < 11;
```

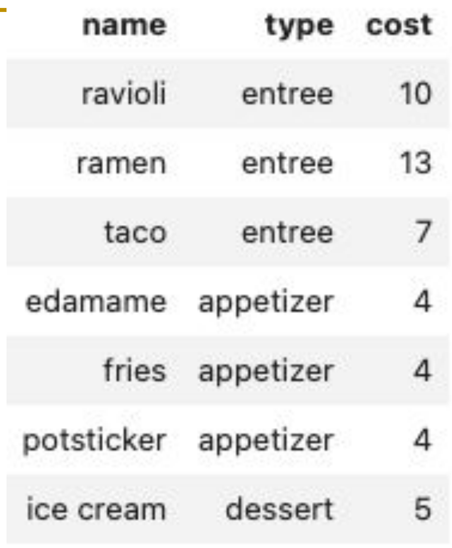


type
entree
appetizer
dessert

```
SELECT DISTINCT type, cost
FROM Dish
WHERE cost < 11;
```



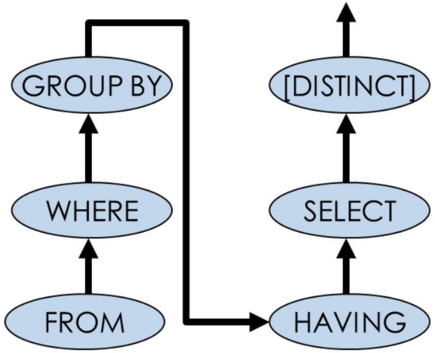
type	cost
entree	10
entree	7
appetizer	4
dessert	5



name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

What does the DISTINCT keyword do? Let's use the flowchart to trace each query.



DISTINCT: What does this do?

```
SELECT DISTINCT type
FROM Dish
WHERE cost < 11;
```

type
entree
appetizer
dessert

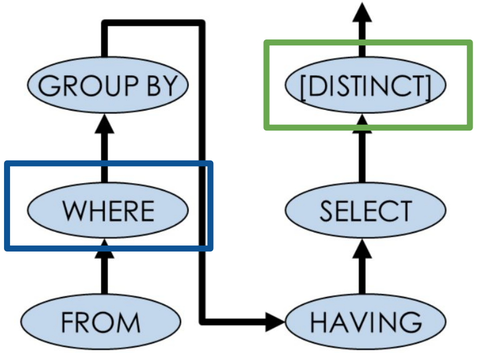
```
SELECT DISTINCT type, cost
FROM Dish
WHERE cost < 11;
```

type	cost
entree	10
entree	7
appetizer	4
dessert	5

name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

Dish

1799583



DISTINCT: What does this do?

```
SELECT DISTINCT type
FROM Dish
WHERE cost < 11;
```

type
entree
appetizer
dessert

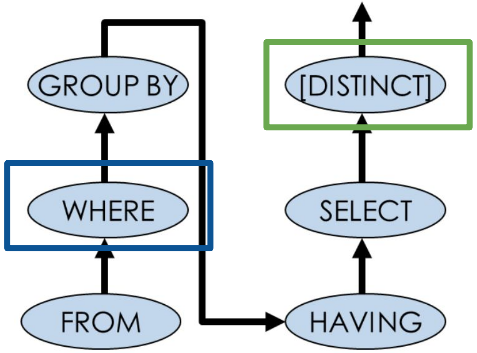
```
SELECT DISTINCT type, cost
FROM Dish
WHERE cost < 11;
```

type	cost
entree	10
entree	7
appetizer	4
dessert	5

name	type	cost
ravioli	entree	10
ramen	entree	13
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

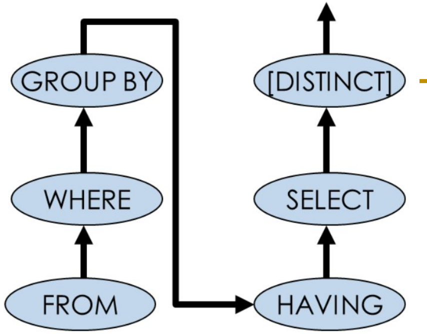
Dish

1799583



DISTINCT creates unique tuples.
WHERE precedes **DISTINCT**.

DISTINCT vs. GROUP BY



1799583

These queries both return unique tuples (ignoring row order) through different precedence of clauses:

```
SELECT DISTINCT
     type,
     cost
FROM Dish;
```

type	cost
entree	10
entree	13
entree	7
appetizer	4
dessert	5

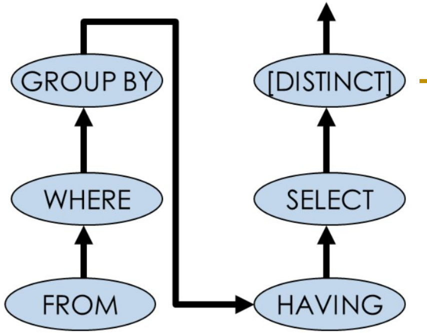
name	type	cost
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

```
SELECT
     type,
     cost
FROM Dish
GROUP BY type, cost;
```

type	cost
appetizer	4
dessert	5
entree	7
entree	10
entree	13

name	type	cost
ravioli	entree	10
pork bun	entree	7
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

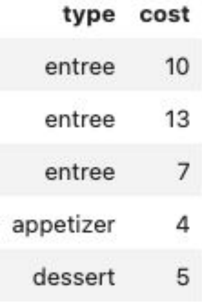
DISTINCT vs. GROUP BY



1799583

These queries both return unique tuples (ignoring row order) through different precedence of clauses:

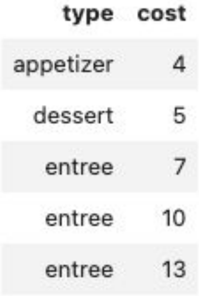
```
SELECT DISTINCT
    type,
    cost
FROM Dish;
```



type	cost
entree	10
entree	13
entree	7
appetizer	4
dessert	5

Better style to use **SELECT DISTINCT** for unique values/tuples.

```
SELECT
    type,
    cost
FROM Dish
GROUP BY type, cost;
```



type	cost
appetizer	4
dessert	5
entree	7
entree	10
entree	13

I think of this as a degenerate use of GROUP BY, because no aggregate functions.

⭐DISTINCT can also be used in Column Expressions!

1799583

Common query: **GROUP BY** and **DISTINCT** **together**.

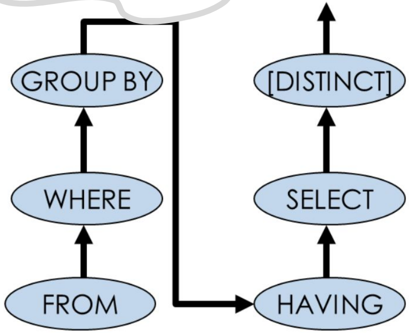
```
SELECT type, AVG(DISTINCT cost)
FROM DishDup
GROUP BY type;
```

type	AVG(DISTINCT cost)
appetizer	4.0
dessert	5.0
entree	8.5

Average of the 7 and 10, which are the unique cost values for entrees.

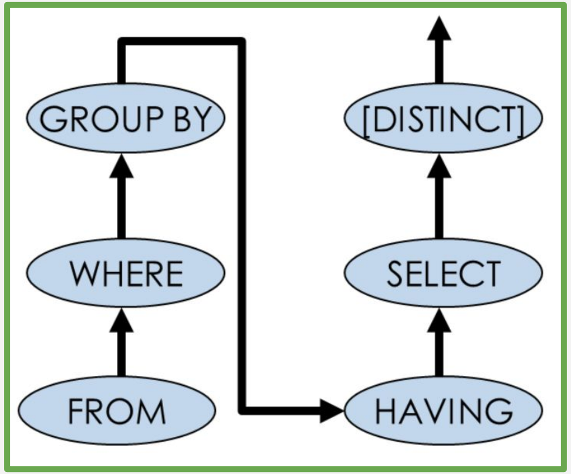
name	type	cost
ravioli	entree	10
ramen	entree	~~13~~ 10
taco	entree	7
edamame	appetizer	4
fries	appetizer	4
potsticker	appetizer	4
ice cream	dessert	5

DishDup



Summary of today (new)

```
SELECT [DISTINCT] <column expression list>
FROM <table>
[WHERE <predicate>]
[GROUP BY <column list>]
[HAVING <predicate>]
[ORDER BY <column list>]
[LIMIT <number of rows>]
[OFFSET <number of rows>];
```



- By convention, use **all caps** for keywords in SQL statements.
- Use **newlines** to make SQL code more readable.
- **AS** keyword: rename columns during selection process.
- `WHERE`: rows; `HAVING`: groups. `WHERE` precedes `HAVING`.
- Column Expressions may include aggregation functions (`MAX`, `MIN`, etc.) **and `DISTINCT`.**

**Got to here.
Will cover on
Thursday 4/6**

Python-SQL

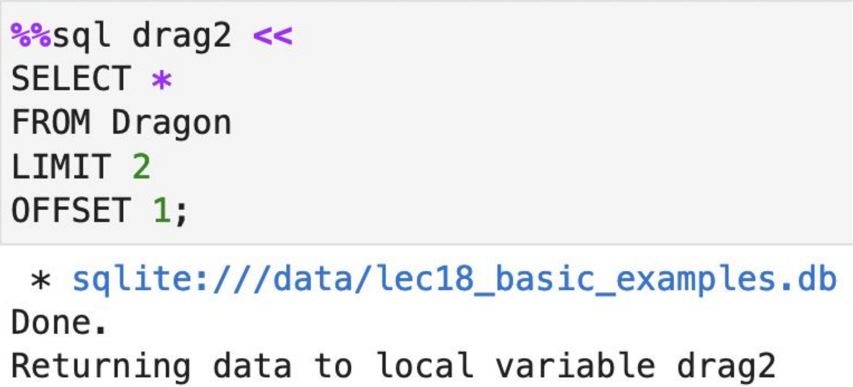
Lecture 21, Data 100 Spring 2023

- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- Filtering Groups with HAVING
- DISTINCT
- **Python SQL**

Python-SQL: from SQL to Python

1799583

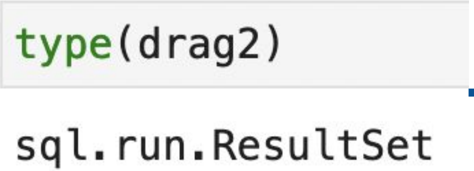
You can store the result of a query into a Python variable, using the << syntax:



```
%%sql drag2 <<
SELECT *
FROM Dragon
LIMIT 2
OFFSET 1;
```

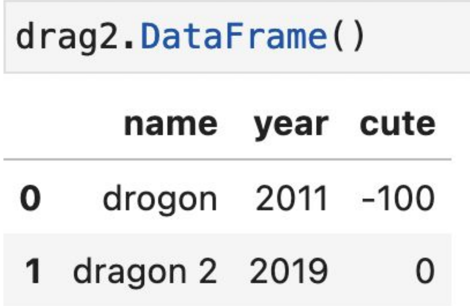
```
 * sqlite:///data/lec18_basic_examples.db
Done.
Returning data to local variable drag2
```

This SQL result can be made into a Pandas DataFrame:



```
type(drag2)
```

```
sql.run.ResultSet
```

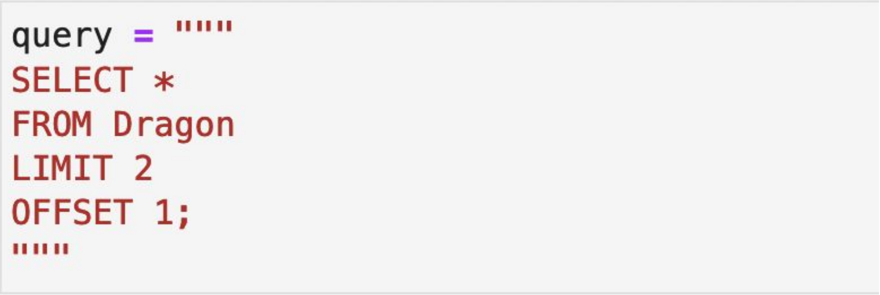


```
drag2.DataFrame()
```

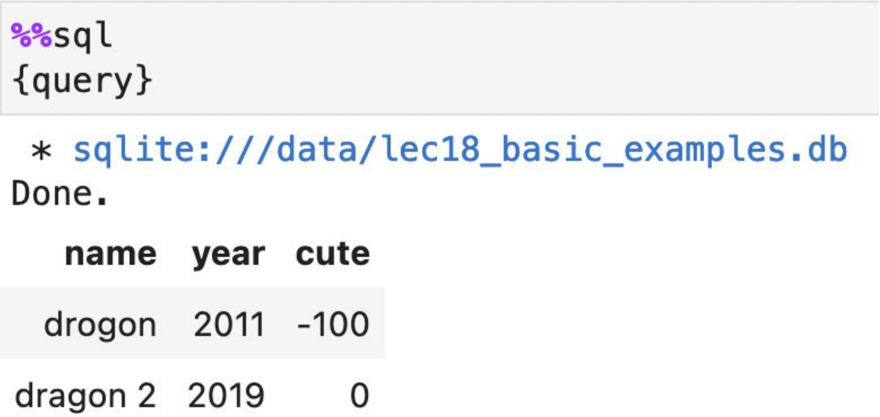
	name	year	cute
0	drogon	2011	-100
1	dragon 2	2019	0

Python-SQL: the other way around

Both of these syntaxes work and let you use Python variables in your SQL magics:

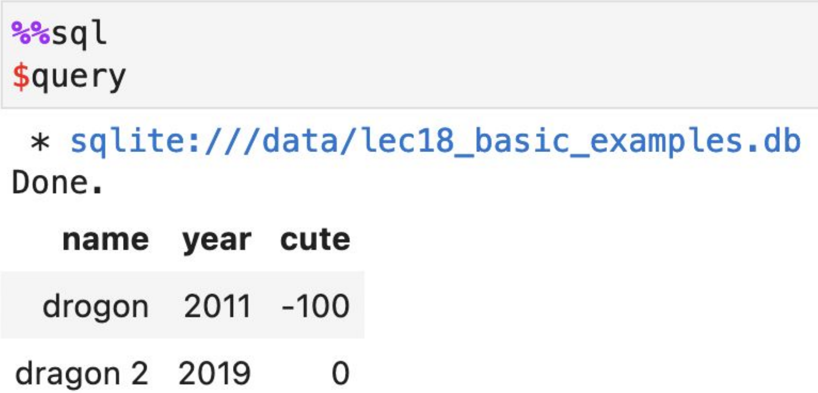


```
query = """
SELECT *
FROM Dragon
LIMIT 2
OFFSET 1;
"""
```



```
%%sql
{query}
```

```
 * sqlite:///data/lec18_basic_examples.db
Done.
   name    year   cute
  drogon   2011   -100
dragon 2   2019      0
```



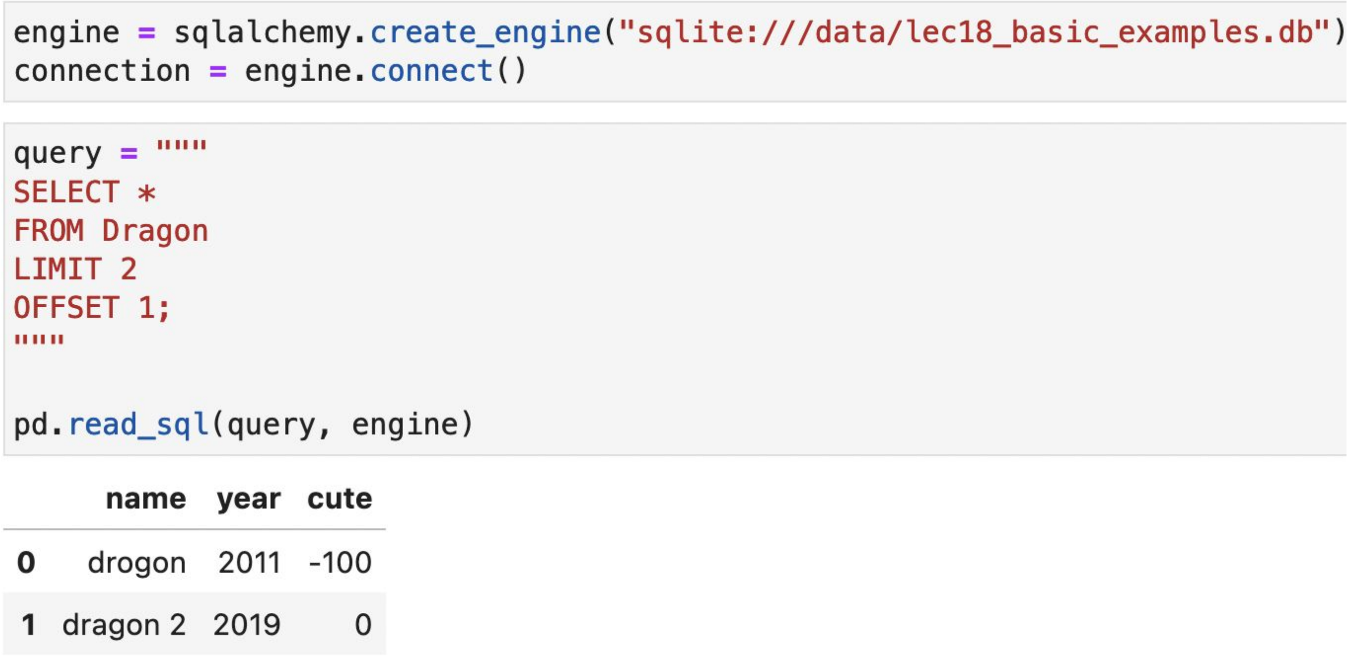
```
%%sql
$query
```

```
 * sqlite:///data/lec18_basic_examples.db
Done.
   name    year   cute
  drogon   2011   -100
dragon 2   2019      0
```

Pandas knows SQL!

1799583

Pandas knows how to talk directly to SQL engines too! You can use whichever syntax you find most convenient:



```
engine = sqlalchemy.create_engine("sqlite:///data/lec18_basic_examples.db")
connection = engine.connect()
```

```
query = """
SELECT *
FROM Dragon
LIMIT 2
OFFSET 1;
"""

pd.read_sql(query, engine)
```

	name	year	cute
0	drogon	2011	-100
1	dragon 2	2019	0

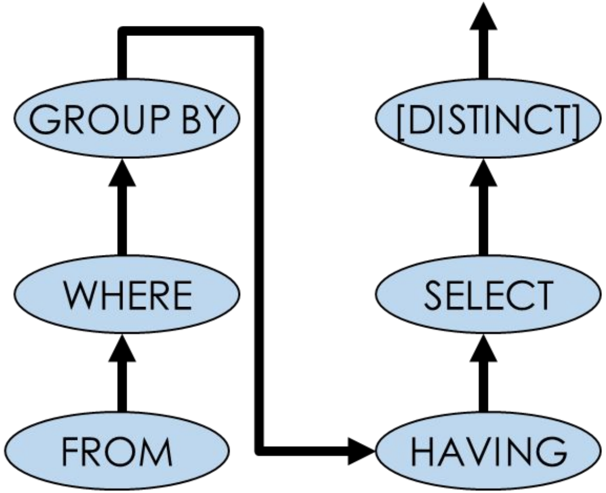
Extra SQL
practice…

Extra Slides (Sp19))

Lecture 21, Data 100 Spring 2023

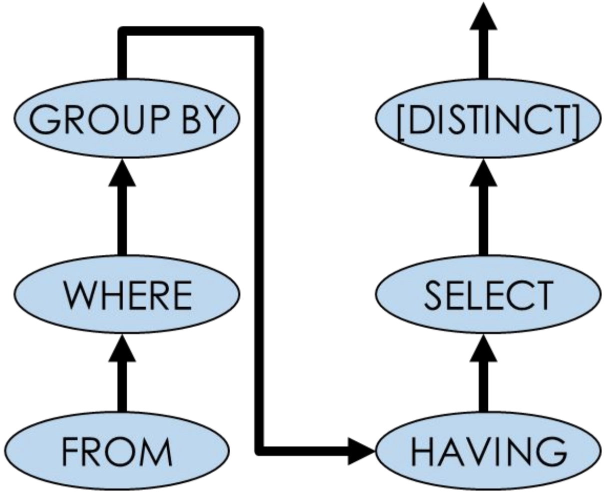
- Why Databases
- Warmup: SQL Example
- SQL Tables
- Basic SQL Queries
- Basic GROUP BY Operations
- Trickier GROUP BY Operations
- DISTINCT
- **Extra Slides**

```
SELECT dept, AVG(gpa) AS avg_gpa, COUNT(*) AS size
  FROM students
 WHERE gender = 'F'
 GROUP BY dept
HAVING COUNT(*) > 2
 ORDER BY avg_gpa DESC
```



What does this compute?

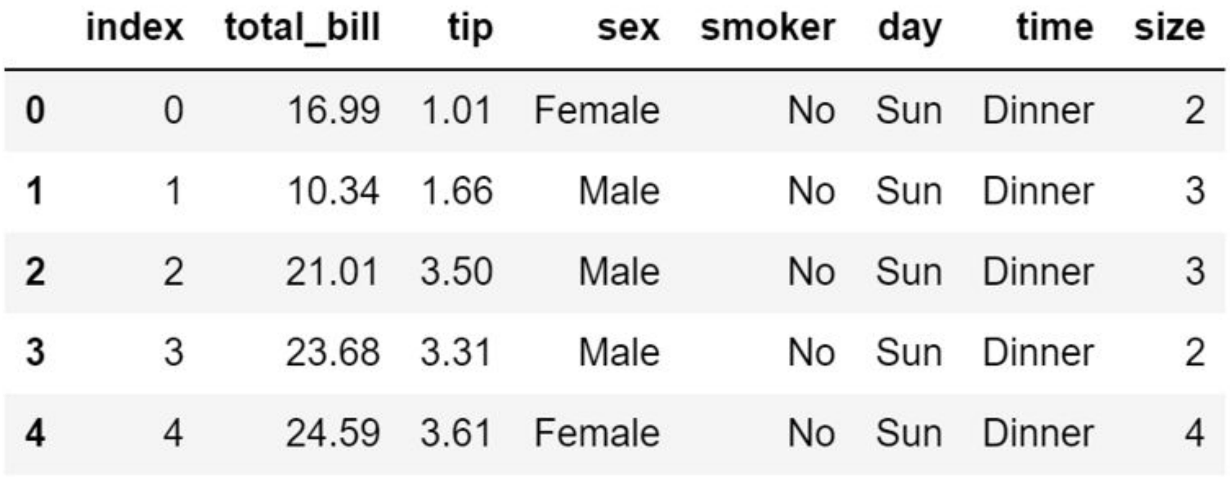
```
SELECT dept, AVG(gpa) AS avg_gpa, COUNT(*) AS size
  FROM students
 WHERE gender = 'F'
 GROUP BY dept
HAVING COUNT(*) > 2
 ORDER BY avg_gpa DESC
```



What does this compute?

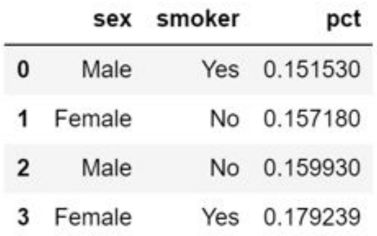
- The average GPA of female students and number of female students in each department where there are at least 3 female students in that department. The results are ordered by the average GPA.

```
SELECT ????
  FROM tips
 WHERE ????
 GROUP BY ????
HAVING ????
 ORDER BY ????
```



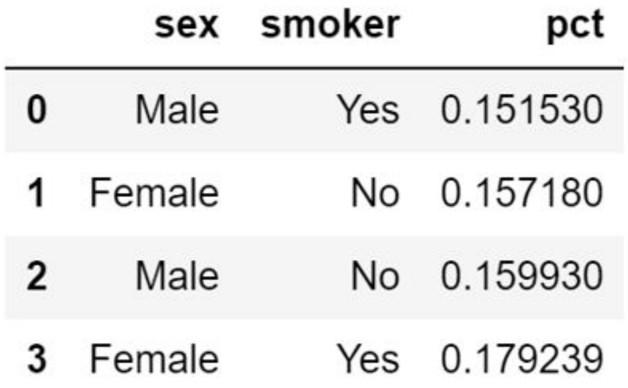
	index	total_bill	tip	sex	smoker	day	time	size
0	0	16.99	1.01	Female	No	Sun	Dinner	2
1	1	10.34	1.66	Male	No	Sun	Dinner	3
2	2	21.01	3.50	Male	No	Sun	Dinner	3
3	3	23.68	3.31	Male	No	Sun	Dinner	2
4	4	24.59	3.61	Female	No	Sun	Dinner	4

Suppose we want to compare smoker vs. non-smoker and female vs. male tips for weekend diners. Create a table ordered by percentage tip that gives the average tip for all four possibilities.



	sex	smoker	pct
0	Male	Yes	0.151530
1	Female	No	0.157180
2	Male	No	0.159930
3	Female	Yes	0.179239

```
SELECT sex, smoker, avg(tip/total_bill) as pct
  FROM tips
 WHERE day = 'Sun' OR day = 'Sat'
 GROUP BY sex, smoker
~~HAVING~~
 ORDER BY pct
```



	sex	smoker	pct
0	Male	Yes	0.151530
1	Female	No	0.157180
2	Male	No	0.159930
3	Female	Yes	0.179239

Suppose we want to compare smoker vs. non-smoker and female vs. male tips for weekend diners. Create a table ordered by percentage tip that gives the average tip for all four possibilities.

LECTURE 21

SQL I

Content credit: Acknowledgments