SQL – part I

Introduction to Data Science
Spring 1404

Yadollah Yaghoobzadeh

Slides are adopted from Data 100/Data 200, Fall 2023 @ UC Berkeley

Goals for Today's Lecture

Stepping away from Python and pandas

- Recognizing situations where we need "bigger" tools for manipulating data
- Writing our first database queries

Agenda

- Why Databases?
- Intro to SQL
- Tables and Schema
- Basic Queries
- Grouping

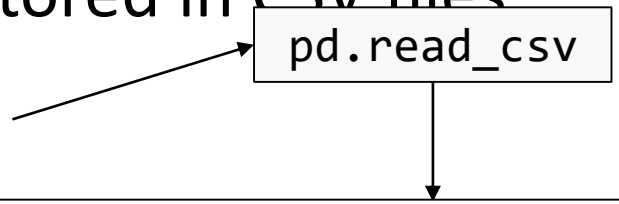
Why Databases?

- **Why Databases?**
- Intro to SQL
- Tables and Schema
- Basic queries
- Grouping

So Far: CSV Files and pandas

- So far, we've worked with data stored in CSV files

`Berkeley_PD_-_Calls_for_Service.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.

Advantages of DBMS over CSV (or Similar)

Data Storage:

- **Reliable storage** to survive system crashes and disk failures.
- Optimize to **compute on data that does not fit in memory**.

Data Management:

- Configure how data is **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).

Intro to SQL

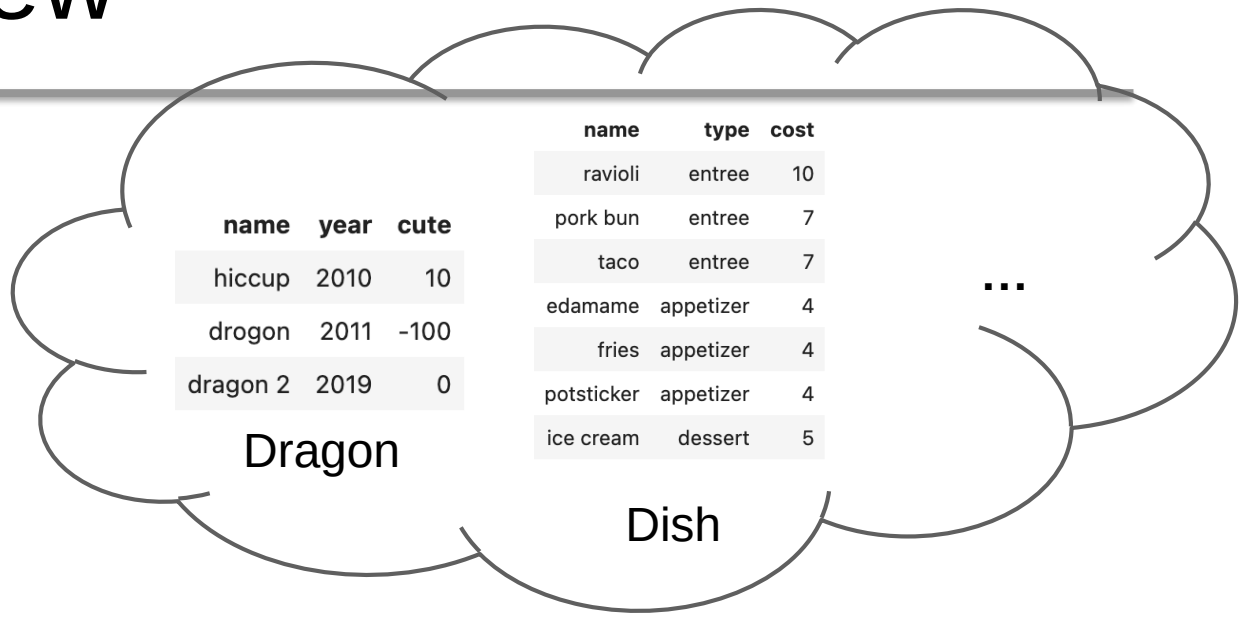
- Why Databases?
- **Intro to SQL**
- Tables and Schema
- Basic queries
- Grouping

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.

How to pronounce? An ongoing debate.

Quick SQL Overview



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

...

SQLite database

SQL query

```
SELECT *
FROM Dish;
```


results

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

Tables and Schema

- Why Databases?
- Intro to SQL
- **Tables and Schema**
- Basic queries
- Grouping

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

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 Schema

A **schema** describes the logical structure of a table. Whenever a new table is created, the creator must declare its schema.

For each column, specify the:

- **Column name**
- **Data type**
- **Constraint(s) on values**

```
CREATE TABLE Dragon (
name TEXT PRIMARY KEY,
year INTEGER CHECK (year >= 2000),
cute INTEGER
)
```

Repeat for all tables in the database:

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))

Example Types

Some examples of SQL **types**:

- INT: Integers.
- FLOAT: Floating point 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

Some examples of **constraints**:

- CHECK: data must obey the given check constraint.
- PRIMARY KEY: specifies that this key is used to uniquely identify rows in the table.
- NOT NULL: null data
- DEFAULT: provides a insertion.

What is this primary key constraint?

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')))

Primary Keys

A **primary key** is the set of column(s) 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…

Basic Queries

- Why Databases?
- Intro to SQL
- Tables and Schema
- **Basic Queries**
- Grouping

Query Syntax So Far

SELECT <*column list*>
FROM <*table*>*;*

; Marks the end of a SQL statement.

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

name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0
puff	2010	100
smaug	2011	None

- Recall our simplest query, which returns the full relation:

```
SELECT *
FROM Dragon;
```

- SELECT specifies the column(s) that we wish to appear in the output. FROM specifies the database table from which to select data.

- *Every* query must include a SELECT clause (how else would we know what to return?) and a FROM clause (how else would we know where to get the data?)

- An asterisk (*) is shorthand for “all columns”.

But first, more SELECT

Recall our simplest query, which returns the full relati

```
SELECT *
FROM Dragon;
```

table name

name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0
puff	2010	100
smaug	2011	None

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
100	2010
None	2011

Columns selected in specified order

Aliasing with AS

To rename a SELECTed column, use the AS keyword

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

cuteness	birth
10	2010
-100	2011
0	2019
100	2010
None	2011

An **alias** is a name given to a column or table by a programmer. Here, “cuteness” is an alias of the original “cute” column (and “birth” is an alias of “year”)

SQL Style: Newline Separators

The following two queries both retrieve the same relation:

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

(more readable)

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

cuteness	birth
10	2010
-100	2011
0	2019
100	2010
None	2011

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

Uniqueness with `DISTINCT`

To return only unique values, combine `SELECT` with the `DISTINCT` keyword

```
SELECT DISTINCT year
FROM Dragon;
```

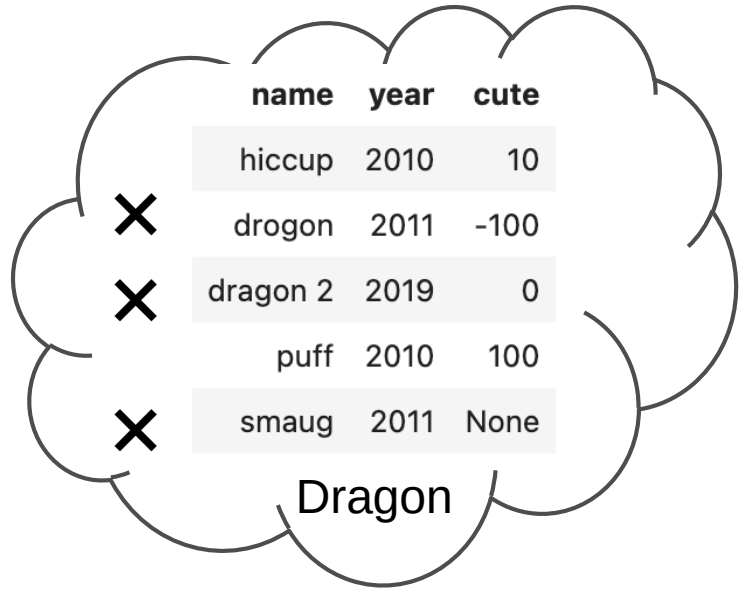
name	year	cute
hiccup	2010	10
drogon	2011	-100
dragon 2	2019	0
puff	2010	100
smaug	2011	None

year
2010
2011
2019

Notice that 2010 and 2011 only appear once each in the output.

WHERE: Select a rows based on conditions

- 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
puff	2010

WHERE: Select a rows based on conditions

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

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

condition

name	cute	year
hiccup	10	2010
dragon 2	0	2019
puff	100	2010

- Check if values are contained IN a specified list

```
SELECT name, year
FROM Dragon
WHERE name IN ('puff', 'hiccup');
```

name	year
hiccup	2010
puff	2010

WHERE with NULL Values

NULL (the SQL equivalent of NaN) is stored in a special format – we can't use the "standard" operators =, >, and <.

Instead, check if something `IS` or `IS NOT NULL`

```
SELECT name, year
FROM Dragon
WHERE year IS NOT NULL;
```

name	cute
hiccup	10
drogon	-100
dragon 2	0
puff	100

Always work with NULLs using the `IS` operator. NULL cannot work with standard comparisons: in fact, NULL = NULL actually returns False!

ORDER BY: Sort rows

- Specify which column(s) we should order the data by

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

column

(by default, SQL orders by ascending order: **ASC**)

name	year	cute
puff	2010	100
hiccup	2010	10
dragon 2	2019	0
drogon	2011	-100
smaug	2011	None

ORDER BY: Sort rows

Specify which column(s) we should order the data by

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

Can also order by multiple columns (for tiebreaks)

Sorts `year` in ascending order and `cute` in descending order. If you want `year` to be ordered in descending order as well, you need to specify `year DESC, cute DESC;`

name	year	cute
puff	2010	100
hiccup	2010	10
drogon	2011	-100
smaug	2011	None
dragon 2	2019	0

OFFSET and LIMIT?

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

Dragon

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.

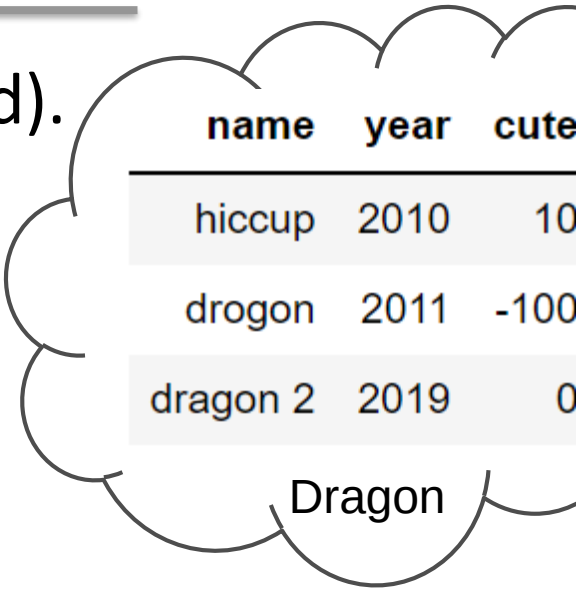
name	year	cute
drogon	2011	-100
dragon 2	2019	0

OFFSET and LIMIT

- 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 tells SQL to skip the first N rows of the output, then apply `LIMIT`.

```
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!

Grouping

- Why Databases?
- Intro to SQL
- Tables and Schema
- Basic Queries
- **Grouping**

The Dish Table

```
SELECT *
FROM Dish;
```

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

The Dish Table

```
SELECT *
FROM Dish;
```

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

Notice the repeated dish types. What if we wanted to investigate trends across each group?

Declarative Programming

Order of operations: SELECT → FROM → WHERE → GROUP BY

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

Correct! ✓

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

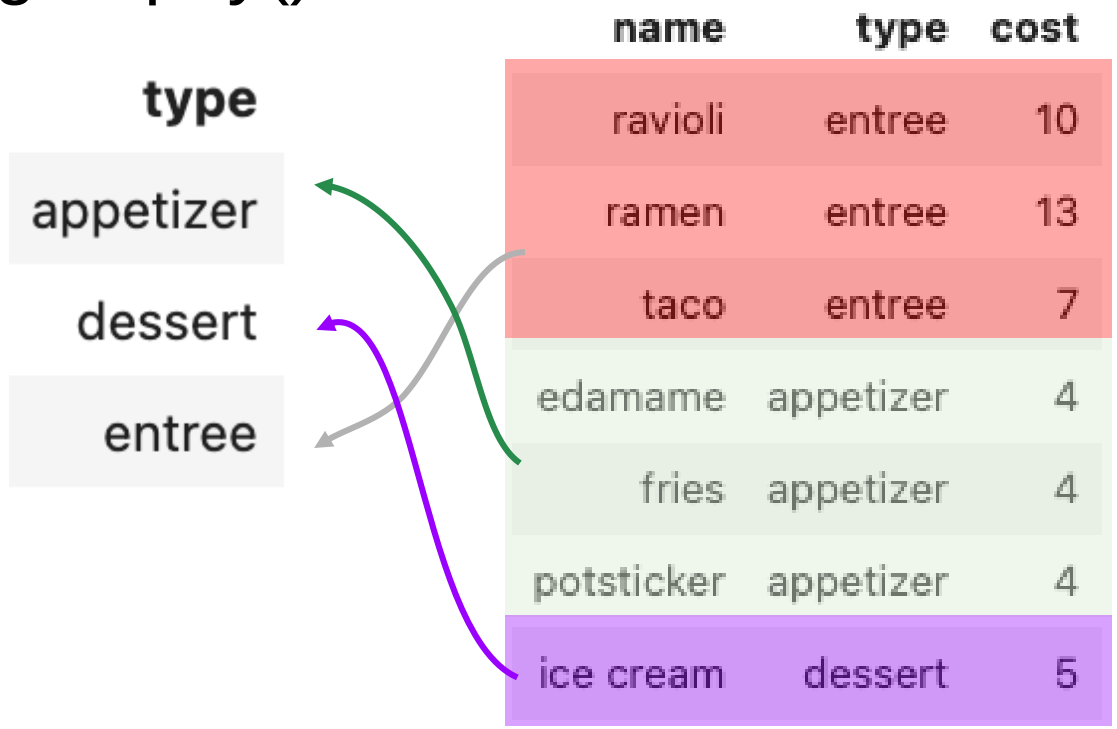
Incorrect ×

Always follow the SQL order of operations. Let SQL take care of the rest.

GROUP BY

GROUP BY is similar to pandas groupby().

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



Aggregating Across Groups

Like `pandas`, SQL has **aggregate functions**: `MAX, SUM, AVG, FIRST,` etc.

For more aggregations, see: https://www.sqlite.org/lang_aggfunc.html

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

type	SUM(cost)
appetizer	12
dessert	5
entree	30

Wait, something's weird…

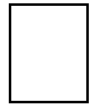
Using Multiple Aggregation Functions

```
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



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

This was much more difficult in pandas!

The COUNT Aggregation

COUNT is used to count the number of rows belonging to a group.

```
SELECT year, COUNT(cute)
FROM Dragon
GROUP BY year;
```

Similar to `pandas groupby().count()`

year	COUNT(cute)
2010	2
2011	1
2019	1

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

```
SELECT year, COUNT(*)
FROM Dragon
GROUP BY year;
```

Similar to `pandas groupby().size()`

year	COUNT(*)
2010	2
2011	2
2019	1