Data Wrangling: Merging and Working with Multiple Data Tables

Nicholas Mattei, *Tulane University*

CMPS3660 – Introduction to Data Science – Fall 2019

https://rebrand.ly/TUDataScience

Many Thanks

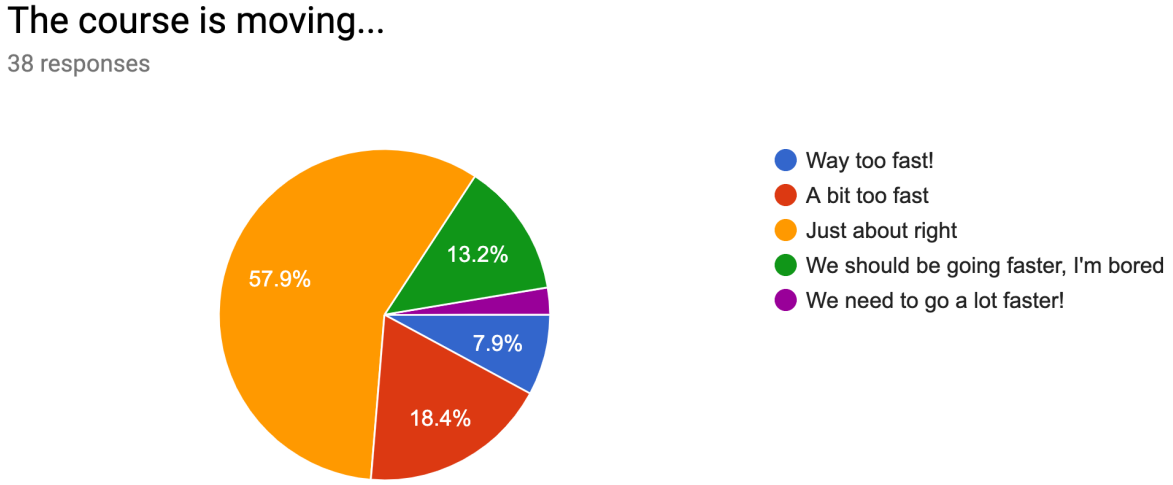
Slides based off Introduction to Data Science from John P. Dickerson - https://cmsc320.github.io/

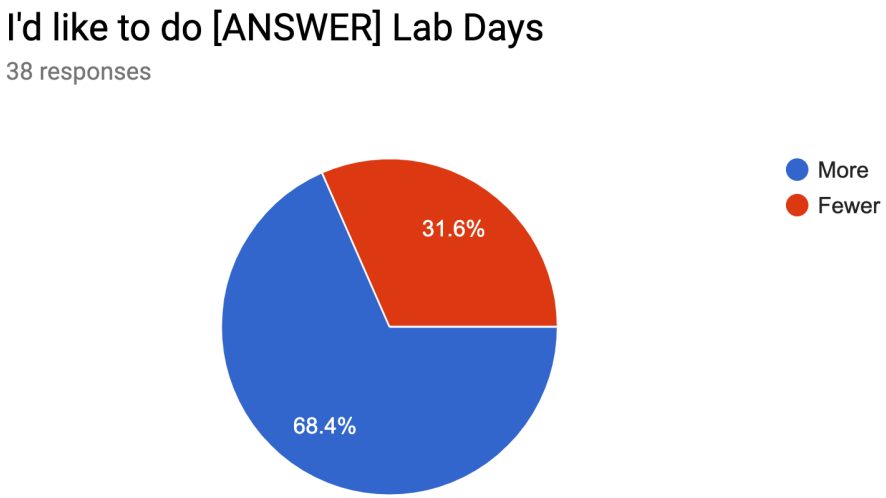
Announcements

- Survey Results!
- Lab 4 + Lab 5
- Weekly Questions 4
- On to DATA!

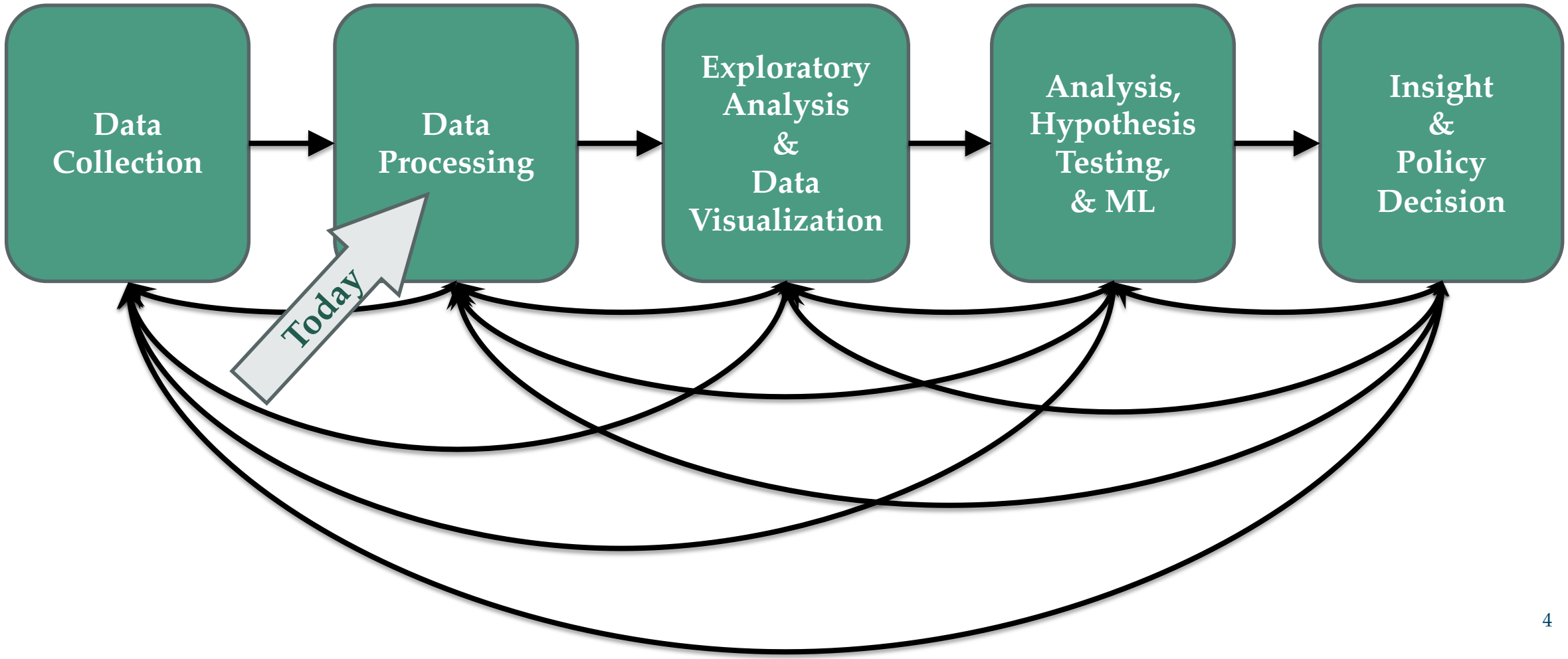
Survey Take Home Messages

- 28+ of you like or really like notebooks!
  - 2 of you really hate them!
- Some of you say I'm talking too fast!
- 6-7 of you Hate Docker!
- What we want:
  - More Depth (2)
  - PPT/Lectures (4)?
    - Too much, Not enough; interesting, boring
  - More Feedback!





The Data LifeCycle



Tables

Special Column, called "Index", or "ID", or "Key"
Usually, no duplicates Allowed

Variables
(also called Attributes, or Columns, or Labels)

Observations, Rows, or Tuples

ID	age	wgt_kg	hgt_cm
1	12.2	42.3	145.1
2	11.0	40.8	143.8
3	15.6	65.3	165.3
4	35.1	84.2	185.8

1. Select/Slicing

- Select only some of the rows, or some of the columns, or a combination.

ID	age	wgt_kg	hgt_cm
1	12.2	42.3	145.1
2	11.0	40.8	143.8
3	15.6	65.3	165.3
4	35.1	84.2	185.8

Only columns ID and Age

ID	age
1	12.2
2	11.0
3	15.6
4	35.1

Only rows with wgt > 41

ID	age	wgt_kg	hgt_cm
1	12.2	42.3	145.1
3	15.6	65.3	165.3
4	35.1	84.2	185.8

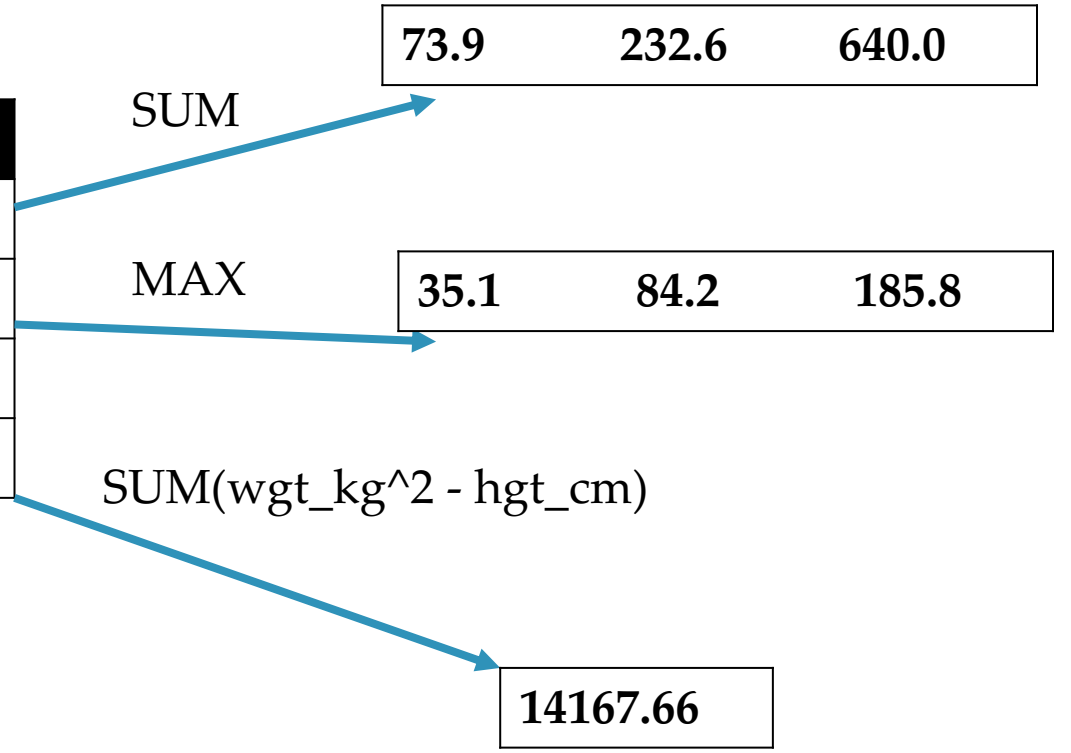
Both

ID	age
1	12.2
3	15.6
4	35.1

2. Aggregate/Reduce

- Combine values across a column into a single value

ID	age	wgt_kg	hgt_cm
1	12.2	42.3	145.1
2	11.0	40.8	143.8
3	15.6	65.3	165.3
4	35.1	84.2	185.8



What about ID/Index column?

Usually not meaningful to aggregate across it

May need to explicitly add an ID column

3. Map

- Apply a function to every row, possibly creating more or fewer columns

ID	Address
1	College Park, MD, 20742
2	Washington, DC, 20001
3	Silver Spring, MD 20901

SPLIT(“,”)

ID	City	State	Zipcode
1	College Park	MD	20742
2	Washington	DC	20001
3	Silver Spring	MD	20901

Variations that allow one row to generate multiple rows in the output (sometimes called “flatmap” or “melt” as we’ll see later.)

4. Group By

- Group tuples together by column/dimension.

ID	A	B	C
1	foo	3	6.6
2	bar	2	4.7
3	foo	4	3.1
4	foo	3	8.0
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

A = foo

ID	B	C
1	3	6.6
3	4	3.1
4	3	8.0
7	4	2.3
8	3	8.0

A = bar

ID	B	C
2	2	4.7
5	1	1.2
6	2	2.5

4. Group By

- Group tuples together by column/dimension.

ID	A	B	C
1	foo	3	6.6
2	bar	2	4.7
3	foo	4	3.1
4	foo	3	8.0
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

B = 1

ID	A	C
5	bar	1.2

B = 2

ID	A	C
2	bar	4.7
6	bar	2.5

B = 3

ID	A	C
1	foo	6.6
4	foo	8.0
8	foo	8.0

B = 4

ID	A	C
3	foo	3.1
7	foo	2.3

4. Group By

- Group tuples together by column/dimension.

ID	A	B	C
1	foo	3	6.6
2	bar	2	4.7
3	foo	4	3.1
4	foo	3	8.0
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

By 'A', 'B'

A = bar, B = 1

ID	C
5	1.2

A = bar, B = 2

ID	C
2	4.7
6	2.5

A = foo, B = 3

ID	C
1	6.6
4	8.0
8	8.0

A = foo, B = 4

ID	C
3	3.1
7	2.3

5. Group By Aggregate

- Group the aggregate per group.

ID	A	B	C
1	foo	3	6.6
2	bar	2	4.7
3	foo	4	3.1
4	foo	3	8.0
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

Group by 'B'
Sum on C

B = 1

ID	A	C
5	bar	1.2

B = 1

Sum (C)
1.2

B = 2

ID	A	C
2	bar	4.7
6	bar	2.5

B = 2

Sum (C)
7.2

B = 3

ID	A	C
1	foo	6.6
4	foo	8.0
8	foo	8.0

B = 3

Sum (C)
22.6

B = 4

ID	A	C
3	foo	3.1
7	foo	2.3

B = 4

Sum (C)
5.4

5. Group By Aggregate

- Final result usually seen as a table.

ID	A	B	C
1	foo	3	6.6
2	bar	2	4.7
3	foo	4	3.1
4	foo	3	8.0
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

Group by 'B'
Sum on C

B = 1

Sum (C)
1.2

B = 2

Sum (C)
7.2

B = 3

Sum (C)
22.6

B = 4

Sum (C)
5.4

B	SUM(C)
1	1.2
2	7.2
3	22.6
4	5.4

5.5 Pivot Tables (Data Cubes)

- Laying out the possible values of multiple axes and aggregating them.
  - Can have more than two dimensions, need hierarchal indexes (later).

ID	A	B	C
1	foo	3	6.6
2	bar	2	4.7
3	foo	4	3.1
4	foo	3	8.0
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

Index A, Columns B

Values C, Agg=Sum

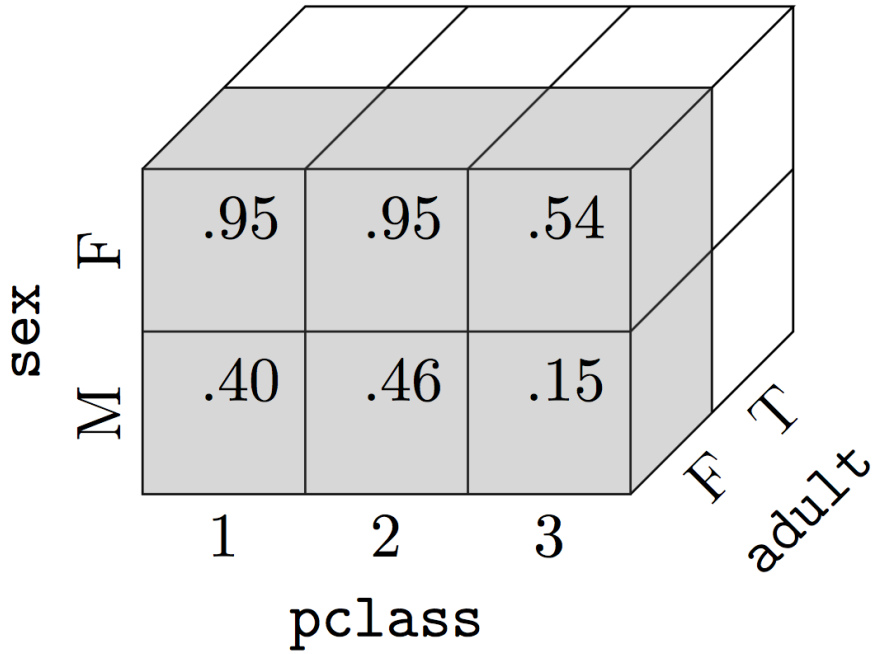
A B>	1	2	3	4
foo	0	0	14.6	5.4
bar	1.2	4.7	0	0

5.5 Pivot Tables (Data Cubes)

- Laying out the possible values of multiple axes and aggregating them.
  - Can have more than two dimensions, need hierarchal indexes (later).

```
1  survivors_cube = titanic_df.pivot_table(
2      index="sex", columns=["adult", "pclass"],
3      values="survived", aggfunc=np.mean)
4  survivors_cube
```

adult	False			True		
pclass	1	2	3	1	2	3
sex						
female	0.947368	0.952381	0.536364	0.968000	0.870588	0.443396
male	0.400000	0.464286	0.147059	0.326389	0.083916	0.155709



6. Union/Intersection/Difference

- Set operations – only if the two tables have identical attributes/columns

ID	A	B	C
1	foo	3	6.6
2	bar	2	4.7
3	foo	4	3.1
4	foo	3	8.0

U

ID	A	B	C
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

→

ID	A	B	C
1	foo	3	6.6
2	bar	2	4.7
3	foo	4	3.1
4	foo	3	8.0
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

Similarly Intersection and Set Difference manipulate tables as Sets

IDs may be treated in different ways, resulting in somewhat different behaviors

7. Merge or Join

- Combine rows/tuples across two tables *if they have the same key.*

ID	A	B
1	foo	3
2	bar	2
3	foo	4
4	foo	3

⋈

ID	C
1	1.2
2	2.5
3	2.3
5	8.0

→

ID	A	B	C
1	foo	3	1.2
2	bar	2	2.5
3	foo	4	2.3

What about IDs not present in both tables?

Often need to keep them around

Can "pad" with NaN (depends on software!)

7. Merge or Join

- Combine rows/tuples across two tables if they have the same key.
- Outer joins can be used to ”pad” IDs that don’t appear in both tables
  - Three variants: LEFT, RIGHT, FULL
  - SQL Terminology -- Pandas has these operations as well

ID	A	B
1	foo	3
2	bar	2
3	foo	4
4	foo	3

ID	C
1	1.2
2	2.5
3	2.3
5	8.0

ID	A	B	C
1	foo	3	1.2
2	bar	2	2.5
3	foo	4	2.3
4	foo	3	NaN
5	NaN	NaN	8.0

Types of Joins

In Pandas this is called a FULL OUTTER JOIN!

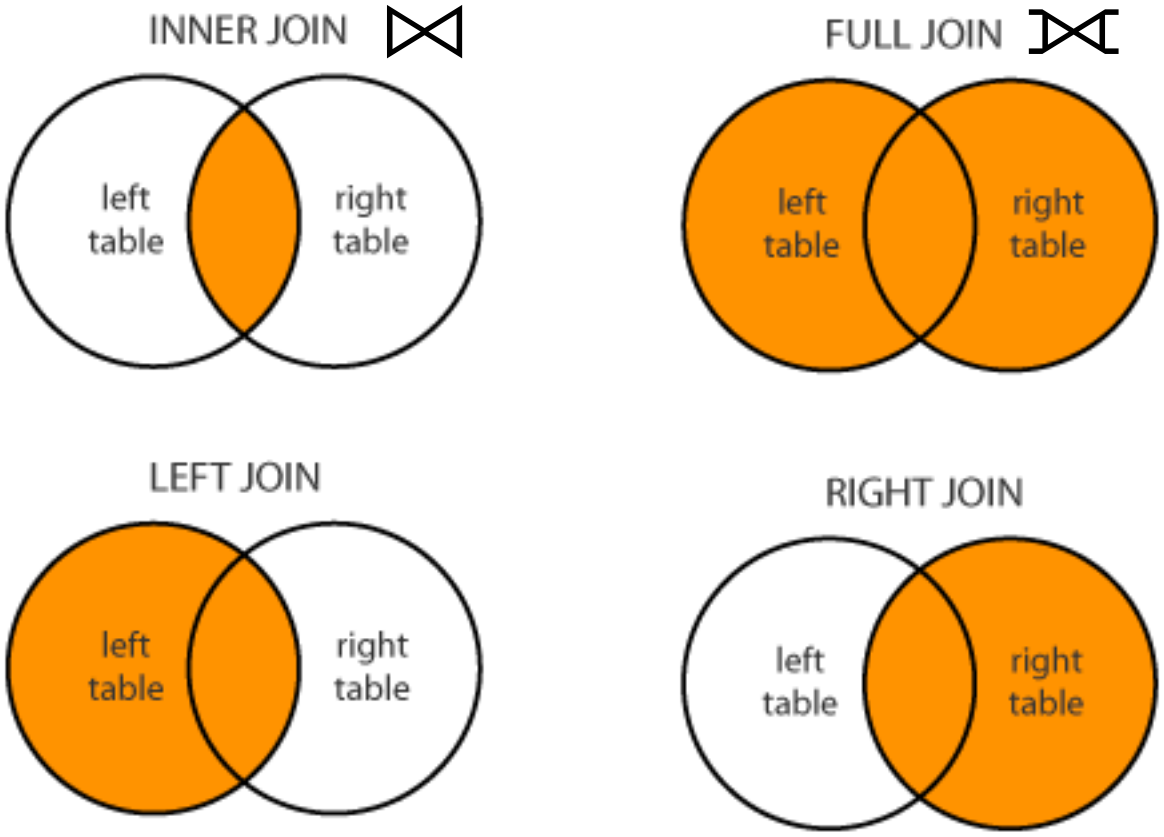


Image credit: http://www.dofactory.com/sql/join

Quick Review

- Tables: A simple, common abstraction
  - Subsumes a set of "strings" – a common input, or a list of lists, or a list of dicts with the same keys.

- Operations on tables:
  - Select, Map, Aggregate, Reduce, Join/Merge, Union/Concat, Group By

- ***These may have different names!*** In Pandas it's a *merge* while in SQL it's a *join*.

- There can be subtle variations in implementation on different data systems. Remember I'm giving you the high level but you need to *read the docs for your software* when you use this stuff!

ID	A	B	C
1	foo	3	6.6
2	baz	2	4.7
3	foo	4	3.1
4	baz	3	8.0
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

Group By 'A'

A = foo

ID	B	C
1	3	6.6
3	4	3.1
7	4	2.3
8	3	8.0

A = baz

ID	B	C
2	2	4.7
4	3	8.0

A = bar

ID	B	C
5	1	1.2
6	2	2.5

How many tuples in the answer?

Note: A GroupBy should partition the whole table!

When does it not?

ID	A	B	C
1	foo	3	6.6
2	baz	2	4.7
3	foo	4	3.1
4	baz	3	8.0
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

Group By ‘A’, ‘B’

A = foo, B=3

ID	C
1	6.6
8	8.0

A = foo, B=4

ID	C
3	3.1
7	2.3

A = baz, B=2

ID	C
2	4.7

A = baz, B=3

ID	C
4	8.0

A = bar, B=1

ID	C
5	1.2

A = bar, B=2

ID	C
6	2.5

How many groups in the answer?

ID	A	B
1	foo	3
2	bar	2
4	foo	4
5	foo	3

$\bowtie$

ID	C
2	1.2
4	2.5
6	2.3
7	8.0

ID	A	B	C
2	bar	2	1.2
4	foo	4	2.5

How many tuples in the answer?

ID	A	B
1	foo	3
2	bar	2
4	foo	4
5	foo	3

⋈

ID	C
2	1.2
4	2.5
6	2.3
7	8.0

→

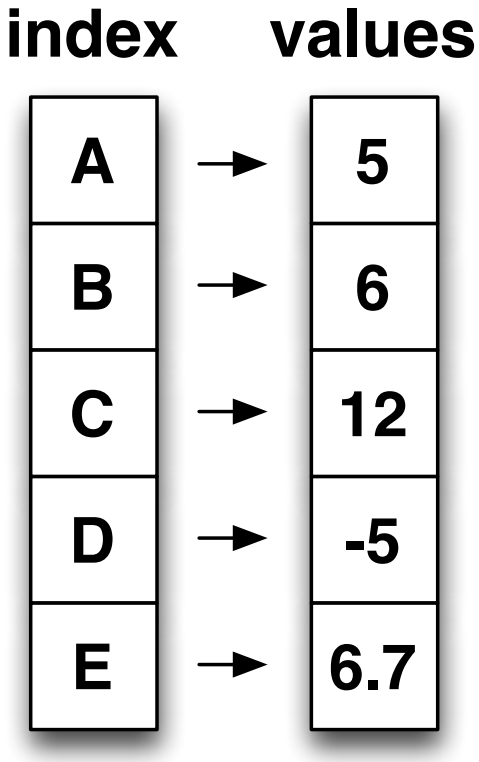
ID	A	B	C
1	foo	3	NaN
2	bar	2	1.2
4	foo	4	2.5
5	foo	3	NaN
6	NaN	NaN	2.3
7	NaN	NaN	8.0

How many tuples in the answer?

Pandas: History

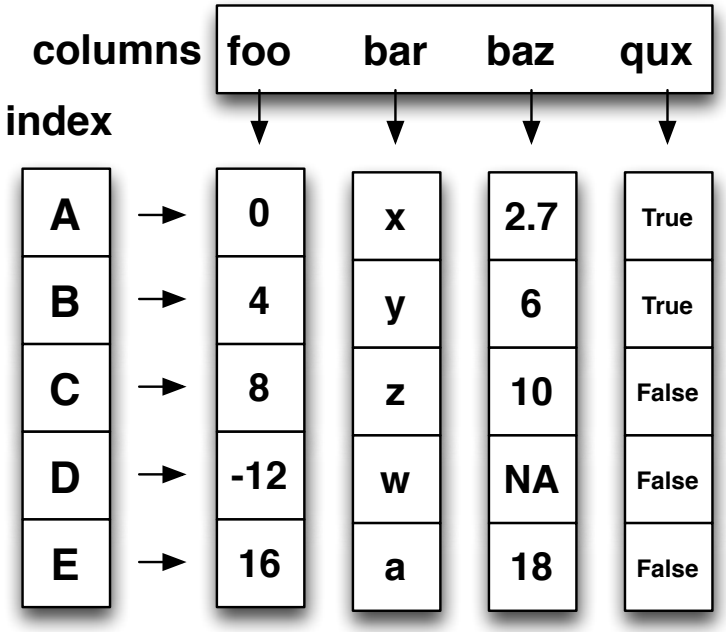
- Written by: Wes McKinney
  - Started in 2008 to get a high-performance, flexible tool to perform quantitative analysis on financial data
- Highly optimized for performance, with critical code paths written in Cython or C
- Key constructs:
  - Series (like a NumPy Array)
  - DataFrame (like a Table or Relation, or R data.frame)
- Foundation for Data Wrangling and Analysis in Python

Pandas: series



- Subclass of numpy.ndarray
- Data: any type
- Index labels need not be ordered
- Duplicates possible but result in reduced functionality

Pandas: DataFrame



- Each column can have a different type
- Row and Column index
- Mutable size: insert and delete columns

- Note the use of word "index" for what we called "key"
  - Relational databases use "index" to mean something else

- Non-unique index values allowed
  - May raise an exception for some operations

Hierarchical Indexes

- Sometimes more intuitive organization of the data
- Makes it easier to understand and analyze higher-dimensional data
- e.g., instead of 3-D array, may only need a 2-D array

day		Fri	Sat	Sun	Thur
sex	smoker				
Female	No	3.125	2.725	3.329	2.460
	Yes	2.683	2.869	3.500	2.990
Male	No	2.500	3.257	3.115	2.942
	Yes	2.741	2.879	3.521	3.058

```
first  second
bar    one       0.469112
       two      -0.282863
baz    one      -1.509059
       two      -1.135632
foo    one       1.212112
       two      -0.173215
qux    one       0.119209
       two      -1.044236
dtype: float64
```

Essential Functionality

- Reindexing to change the index associated with a DataFrame
  - Common usage to interpolate, fill in missing values

```
In [84]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [85]: obj3.reindex(range(6), method='ffill')
Out[85]:
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
```


From: Python for Data Analysis; Wes McKinney

Essential Functionality

- "drop" to delete entire rows or columns
- Indexing, Selection, Filtering: very similar to NumPy
- Arithmetic Operations
  - Result index union of the two input indexes
  - Options to do "fill" while doing these operations

```
In [128]: s1
Out[128]:
a    7.3
c   -2.5
d    3.4
e    1.5
```

```
In [129]: s2
Out[129]:
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
```

```
In [130]: s1 + s2
Out[130]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
```

Function application and Mapping

```
In [158]: frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),
   .....:                   index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
In [159]: frame
Out[159]:
               b         d         e
Utah   -0.204708  0.478943 -0.519439
Ohio   -0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189 -1.296221
```

```
In [160]: np.abs(frame)
Out[160]:
               b         d         e
Utah    0.204708  0.478943  0.519439
Ohio    0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189  1.296221
```

```
In [161]: f = lambda x: x.max() - x.min()
```

```
In [162]: frame.apply(f)
Out[162]:
b    1.802165
d    1.684034
e    2.689627
```

```
In [163]: frame.apply(f, axis=1)
Out[163]:
Utah      0.998382
Ohio      2.521511
Texas     0.676115
Oregon    2.542656
```


From: Python for Data Analysis; Wes McKinney

Sorting and Ranking

```
In [169]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

In [170]: obj.sort_index()
Out[170]:
a    1
b    2
c    3
d    0
```

```
In [187]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
   .....:                    'c': [-2, 5, 8, -2.5]})

In [188]: frame
Out[188]:
   a    b    c
0  0  4.3 -2.0
1  1  7.0  5.0
2  0 -3.0  8.0
3  1  2.0 -2.5
```

```
In [189]: frame.rank(axis=1)
Out[189]:
   a  b  c
0  2  3  1
1  1  3  2
2  2  1  3
3  2  3  1
```


From: Python for Data Analysis; Wes McKinney

Descriptive and Summary Statistics

Table 5-10. Descriptive and summary statistics

Method	Description
`count`	Number of non-NA values
`describe`	Compute set of summary statistics for Series or each DataFrame column
`min, max`	Compute minimum and maximum values
`argmin, argmax`	Compute index locations (integers) at which minimum or maximum value obtained, respectively
`idxmin, idxmax`	Compute index values at which minimum or maximum value obtained, respectively
`quantile`	Compute sample quantile ranging from 0 to 1
`sum`	Sum of values
`mean`	Mean of values
`median`	Arithmetic median (50% quantile) of values
`mad`	Mean absolute deviation from mean value
`var`	Sample variance of values
`std`	Sample standard deviation of values
`skew`	Sample skewness (3rd moment) of values
`kurt`	Sample kurtosis (4th moment) of values
`cumsum`	Cumulative sum of values
`cummin, cummax`	Cumulative minimum or maximum of values, respectively
`cumprod`	Cumulative product of values
`diff`	Compute 1st arithmetic difference (useful for time series)
`pct_change`	Compute percent changes

From: Python for Data Analysis; Wes McKinney

Creating Dataframes

- Directly from Dict or Series
- From a Comma-Separated File – CSV file
  - pandas.read_csv()
  - Can infer headers/column names if present, otherwise may want to reindex
- From an Excel File
  - pandas.read_excel()
- From a Database using SQL (see the reading for an example)
- From Clipboard, URL, Google Analytics, …
- …

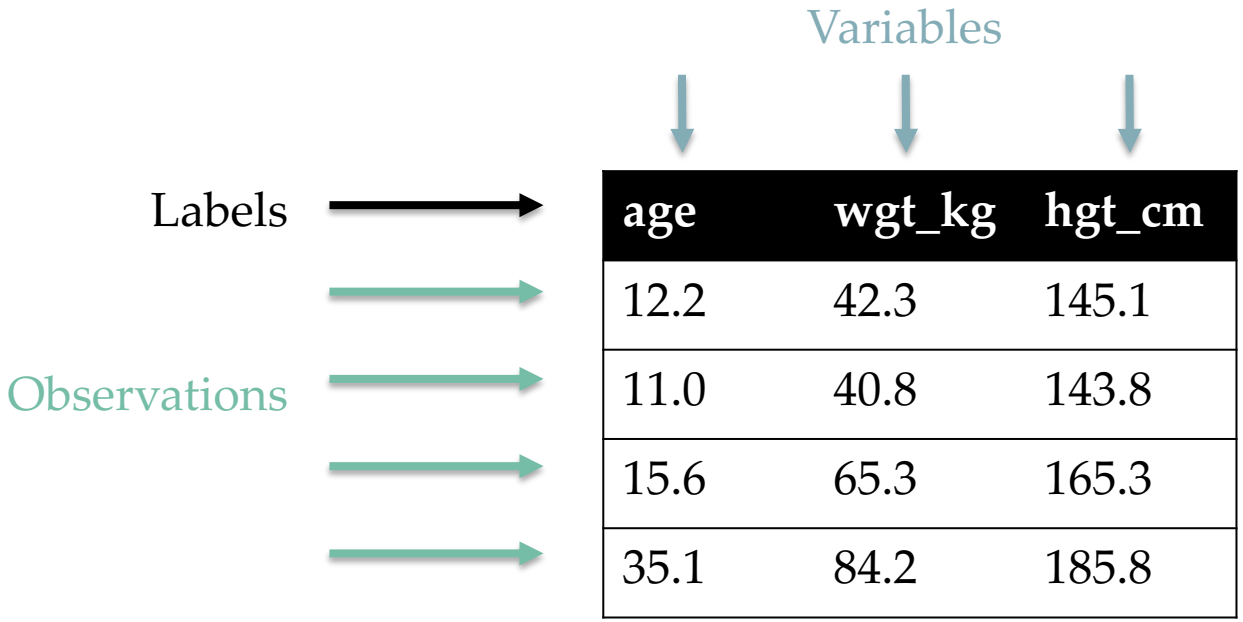
From: Python for Data Analysis; Wes McKinney

More…

- Unique values, Value counts
- Correlation and Covariance
- Functions for handling missing data – in a few classes
  - dropna(), fillna()
- Broadcasting
- Pivoting

- We will see some of these as we discuss data wrangling, cleaning, etc.

Tidy Data



age	wgt_kg	hgt_cm
12.2	42.3	145.1
11.0	40.8	143.8
15.6	65.3	165.3
35.1	84.2	185.8

- But also:
- Names of files/DataFrames = description of one dataset
- Enforce one data type per dataset (ish)

Example

- Variable: measure or attribute:
- age, weight, height, sex
- Value: measurement of attribute:
- 12.2, 42.3kg, 145.1cm, M/F
- Observation: all measurements for an object
- A specific person is [12.2, 42.3, 145.1, F]

Tidying Data I

Name	Treatment A	Treatment B
John Smith	-	2
Jane Doe	16	11
Mary Johnson	3	1

??????????????

Name	Treatment A	Treatment B	Treatment C	Treatment D
John Smith	-	2	-	-
Jane Doe	16	11	4	1
Mary Johnson	3	1	-	2

??????????????

Thanks to http://jeannicholashould.com/tidy-data-in-

Tidying Data II

In a few lectures …

Name	Treatment	Result
John Smith	A	-
John Smith	B	2
John Smith	C	-
John Smith	D	-
Jane Doe	A	16
Jane Doe	B	11
Jane Doe	C	4
Jane Doe	D	1
Mary Johnson	A	3
Mary Johnson	B	1
Mary Johnson	C	-
Mary Johnson	D	2

Melting Data I

Tulane University

religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k	$50-75k
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Dont know/refused	15	14	15	11	10	35
Evangelical Prot	575	869	1064	982	881	1486
Hindu	1	9	7	9	11	34
Historically Black Prot	228	244	236	238	197	223
Jehovahs Witness	20	27	24	24	21	30
Jewish	19	19	25	25	30	95

??????????????

Melting Data II

```
f_df = pd.melt(df,
               ["religion"],
               var_name="income",
               value_name="freq")
f_df = f_df.sort_values(by=["religion"])
f_df.head(10)
```

religion	income	freq
Agnostic	<$10k	27
Agnostic	$30-40k	81
Agnostic	$40-50k	76
Agnostic	$50-75k	137
Agnostic	$10-20k	34
Agnostic	$20-30k	60
Atheist	$40-50k	35
Atheist	$20-30k	37
Atheist	$10-20k	27
Atheist	$30-40k	52

More complicated example

- Billboard Top 100 data for songs, covering their position on the Top 100 for 75 weeks, with two "messy" bits:
- Column headers for each of the 75 weeks
- If a song didn't last 75 weeks, those columns have are null

year	artist.inverted	track	time	genre	date.entered	date.peaked	x1st.week	x2nd.week	...
2000	Destiny's Child	Independent Women Part I	3:38	Rock	2000-09-23	2000-11-18	78	63.0	...
2000	Santana	Maria, Maria	4:18	Rock	2000-02-12	2000-04-08	15	8.0	...
2000	Savage Garden	I Knew I Loved You	4:07	Rock	1999-10-23	2000-01-29	71	48.0	...
2000	Madonna	Music	3:45	Rock	2000-08-12	2000-09-16	41	23.0	...
2000	Aguilera, Christina	Come On Over Baby	3:38	Rock	2000-08-05	2000-10-14	57	47.0	...
2000	Janet	Doesn't Really Matter	4:17	Rock	2000-06-17	2000-08-26	59	52.0	

Messy columns!

Thanks to http://jeannicholashould.com/tidy-data-in-

More complicated example

```
# Keep identifier variables
id_vars = ["year",
           "artist.inverted",
           "track",
           "time",
           "genre",
           "date.entered",
           "date.peaked"]

# Melt the rest into week and rank columns
df = pd.melt(frame=df,
             id_vars=id_vars,
             var_name="week",
             value_name="rank")
```

- Creates one row per week, per record, with its rank

More complicated example

```
# Formatting
df["week"] = df['week'].str.extract('(\d+)',
                                    expand=False).astype(int)
df["rank"] = df["rank"].astype(int)
```

[…, "x2nd.week", 63.0] → […, 2, 63]

```
# Cleaning out unnecessary rows
df = df.dropna()

# Create "date" columns
df['date'] = pd.to_datetime(
                df['date.entered']) +
                pd.to_timedelta(df['week'], unit='w') –
                pd.DateOffset(weeks=1)
```

More complicated example

```
# Ignore now-redundant, messy columns
df = df[["year",
         "artist.inverted",
         "track",
         "time",
         "genre",
         "week",
         "rank",
         "date"]]

df = df.sort_values(ascending=True,
 by=["year","artist.inverted","track","week","rank"])

# Keep tidy dataset for future usage
billboard = df

df.head(10)
```

More complicated example

year	artist.inverted	track	time	genre	week	rank	date
2000	2 Pac	Baby Don't Cry (Keep Ya Head Up II)	4:22	Rap	1	87	2000-02-26
2000	2 Pac	Baby Don't Cry (Keep Ya Head Up II)	4:22	Rap	2	82	2000-03-04
2000	2 Pac	Baby Don't Cry (Keep Ya Head Up II)	4:22	Rap	3	72	2000-03-11
2000	2 Pac	Baby Don't Cry (Keep Ya Head Up II)	4:22	Rap	4	77	2000-03-18
2000	2 Pac	Baby Don't Cry (Keep Ya Head Up II)	4:22	Rap	5	87	2000-03-25
2000	2 Pac	Baby Don't Cry (Keep Ya Head Up II)	4:22	Rap	6	94	2000-04-01
2000	2 Pac	Baby Don't Cry (Keep Ya Head Up II)	4:22	Rap	7	99	2000-04-08
2000	2Ge+her	The Hardest Part Of Breaking Up (Is Getting Ba...	3:15	R&B	1	91	2000-09-02
2000	2Ge+her	The Hardest Part Of Breaking Up (Is Getting Ba...	3:15	R&B	2	87	2000-09-09
2000	2Ge+her	The Hardest Part Of Breaking Up (Is Getting Ba...	3:15	R&B	3	92	2000-09-16

??????????????

More to do?

- Column headers are values, not variable names?
- Good to go!
- Multiple variables are stored in one column?
- Maybe (depends on if genre text in raw data was multiple)
- Variables are stored in both rows and columns?
- Good to go!
- Multiple types of observational units in the same table?
- Good to go! One row per song's week on the Top 100.
- A single observational unit is stored in multiple tables?
- Don't do this!
- Repetition of data?
- Lots! Artist and song title's text names. Which leads us to …

Today/Next Class

- Tables
  - Abstraction
  - Operations
- Pandas
- Tidy Data
- SQL and Relational Databases

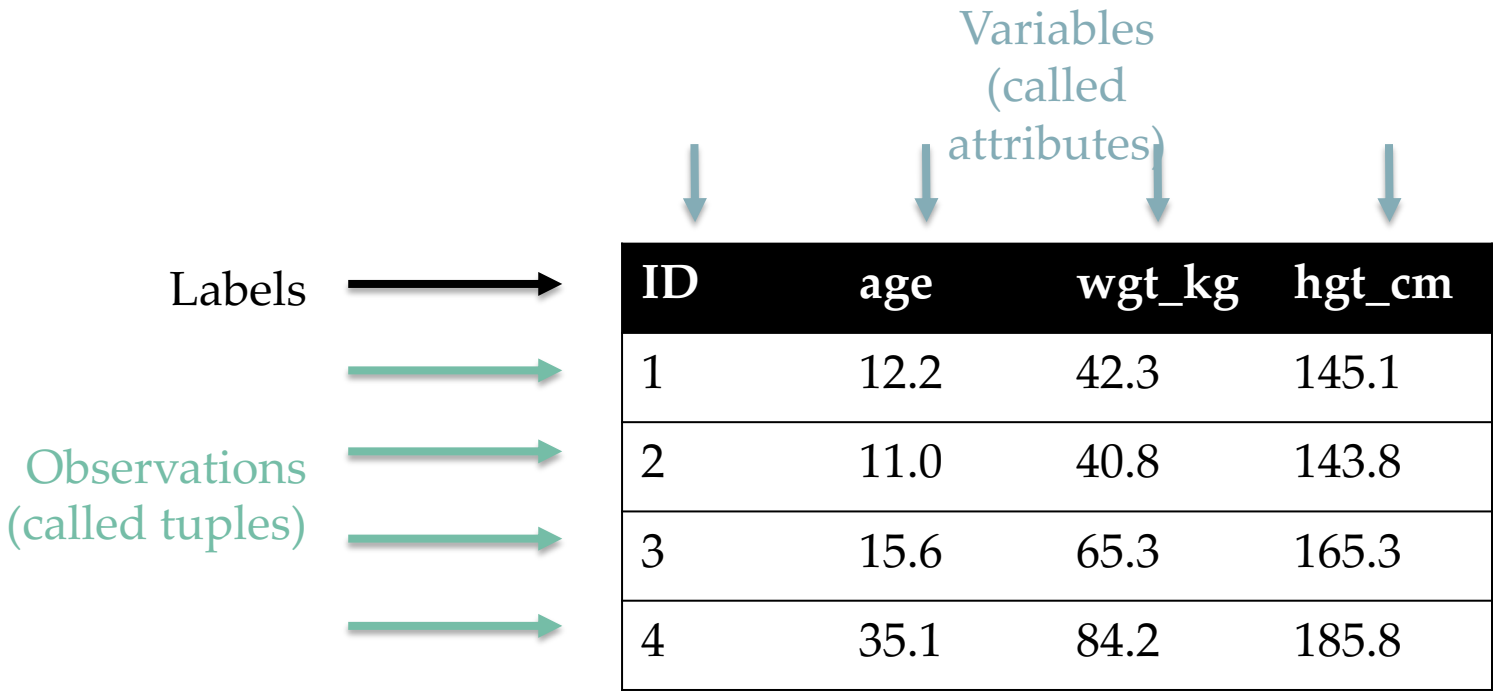
Today's Lecture

- Relational data:
- What is a relation, and how do they interact?
- Querying databases:
- SQL
- SQLite
- How does this relate to `pandas`?
- Joins

Thanks to Zico Kolter for some structure for this lecture!

Relation

- Simplest relation: a table aka tabular data full of unique tuples



ID	age	wgt_kg	hgt_cm
1	12.2	42.3	145.1
2	11.0	40.8	143.8
3	15.6	65.3	165.3
4	35.1	84.2	185.8

Primary keys

ID	age	wgt_kg	hgt_cm	nat_id
1	12.2	42.3	145.1	1
2	11.0	40.8	143.8	1
3	15.6	65.3	165.3	2
4	35.1	84.2	185.8	1
5	18.1	62.2	176.2	3
6	19.6	82.1	180.1	1

ID	Nationality
1	USA
2	Canada
3	Mexico

- The primary key is a unique identifier for every tuple in a relation
- Each tuple has exactly one primary key

Aren't these called "indexes"?

- Yes, in Pandas; but not in the database world
- For most databases, an "index" is a data structure used to speed up retrieval of specific tuples
- For example, to find all tuples with nat_id = 2:
  - We can either scan the table – O(N)
  - Or use an "index" (e.g., binary tree) – O(log N)

Foreign keys

ID	age	wgt_kg	hgt_cm	nat_id
1	12.2	42.3	145.1	1
2	11.0	40.8	143.8	1
3	15.6	65.3	165.3	2
4	35.1	84.2	185.8	1
5	18.1	62.2	176.2	3
6	19.6	82.1	180.1	1

ID	Nationality
1	USA
2	Canada
3	Mexico

- Foreign keys are attributes (columns) that point to a different table's primary key
- A table can have multiple foreign keys

Relation Schema

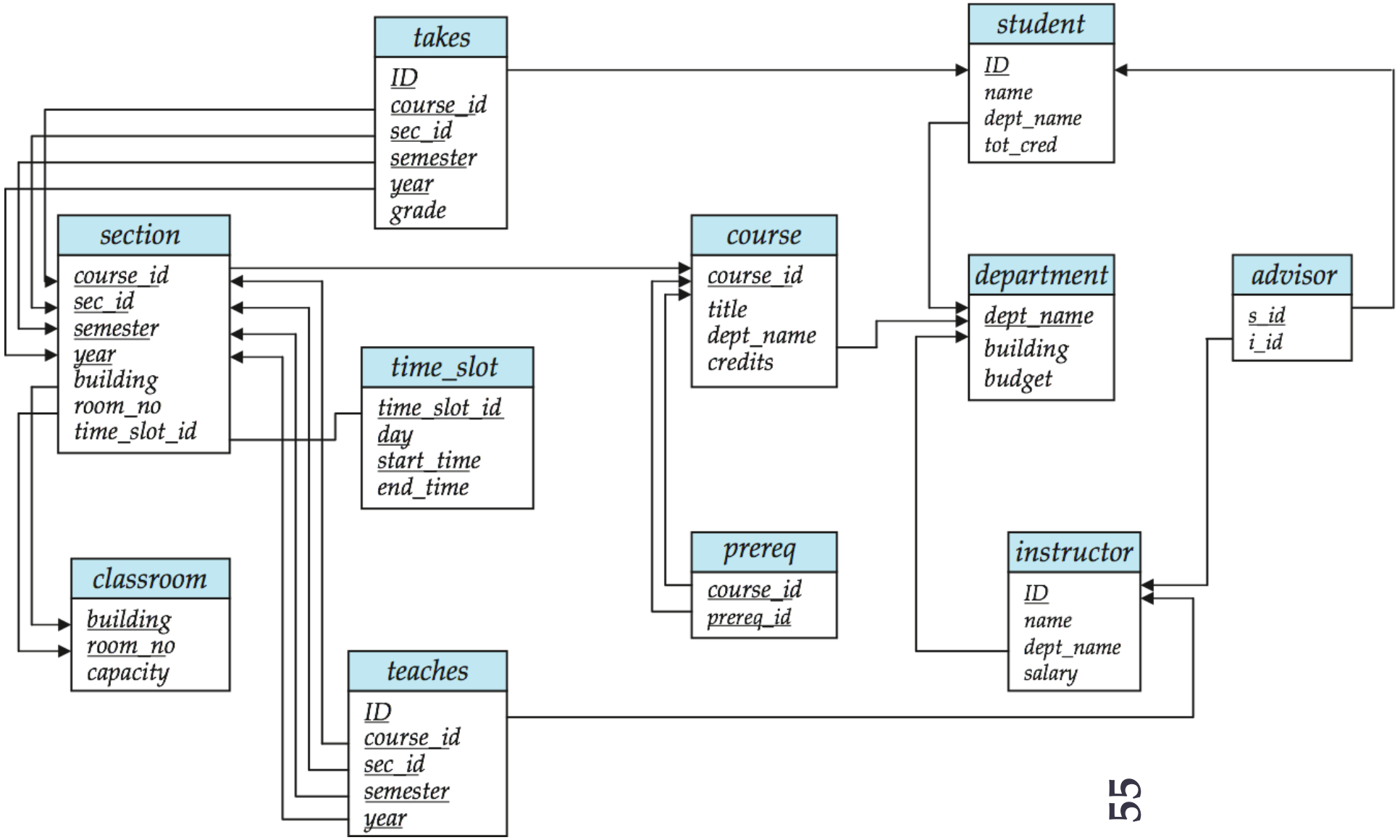
- A list of all the attribute names, and their *domains*

```
create table department
    (dept_name varchar(20),
     building varchar(15),
     budget numeric(12,2) check (budget > 0),
     primary key (dept_name)
    );
```

SQL Statements To create Tables

```
create table instructor (
     ID         char(5),
     name       varchar(20) not null,
     dept_name  varchar(20),
     salary     numeric(8,2),
     primary key (ID),
     foreign key (dept_name) references department
)
```

Schema Diagrams



Searching for elements

- Find all people with nationality Canada (nat_id = 2):
  - ??????????????

ID	age	wgt_kg	hgt_cm	nat_id
1	12.2	42.3	145.1	1
2	11.0	40.8	143.8	1
3	15.6	65.3	165.3	2
4	35.1	84.2	185.8	1
5	18.1	62.2	176.2	3
6	19.6	82.1	180.1	1

O(n) 

Indexes

- Like a hidden sorted map of references to a specific attribute (column) in a table; allows O(log n) lookup instead of O(n)

loc	ID	age	wgt_kg	hgt_cm	nat_id
0	1	12.2	42.3	145.1	1
128	2	11.0	40.8	143.8	2
256	3	15.6	65.3	165.3	2
384	4	35.1	84.2	185.8	1
512	5	18.1	62.2	176.2	3
640	6	19.6	82.1	180.1	1

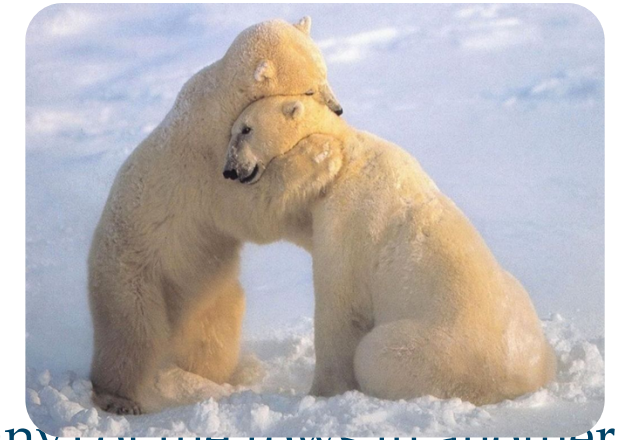
nat_id	locs
1	0, 384, 640
2	128, 256
3	512

INdexes

- Actually implemented with data structures like B-trees
- (Take courses like CMSC424 or CMSC420)
- But: indexes are not free
- Takes memory to store
- Takes time to build
- Takes time to update (add/delete a row, update the column)
- But, but: one index is (mostly) free
- Index will be built automatically on the primary key
- Think before you build/maintain an index on other attributes!

Relationships

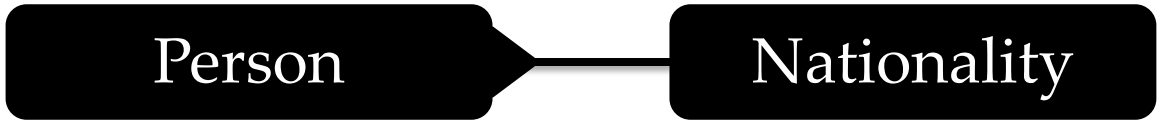
- Primary keys and foreign keys define interactions between different tables aka entities. Four types:
- One-to-one
- One-to-one-or-none
- One-to-many and many-to-one
- Many-to-many



- Connects (one, many) of the rows in one table to (one, many) of the rows in another table

One-to-many & Many-to-one

- One person can have one nationality in this example, but one nationality can include many people.



ID	age	wgt_kg	hgt_cm	nat_id
1	12.2	42.3	145.1	1
2	11.0	40.8	143.8	1
3	15.6	65.3	165.3	2
4	35.1	84.2	185.8	1
5	18.1	62.2	176.2	3
6	19.6	82.1	180.1	1

ID	Nationality
1	USA
2	Canada
3	Mexico

One-to-One

- Two tables have a one-to-one relationship if every tuple in the first tables corresponds to exactly one entry in the other

- In general, you won't be using these (why not just merge the rows into one table?) unless:
- Split a big row between SSD and HDD or distributed
- Restrict access to part of a row (some DBMSs allow column-level access control, but not all)
- Caching, partitioning, & serious stuff: take CMSC424

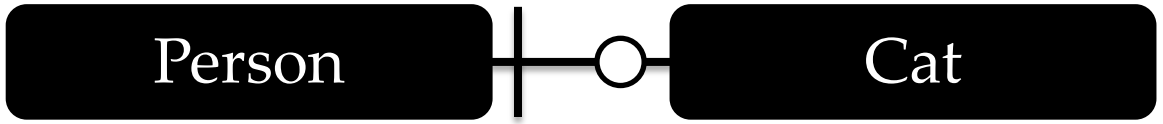
One-to-One-Or-None

- Say we want to keep track of people's cats:

Person ID	Cat1	Cat2
1	Chairman Meow	Fuzz Aldrin
4	Anderson Pooper	Meowly Cyrus
5	Gigabyte	Megabyte

- People with IDs 2 and 3 do not own cats*, and are not in the table. Each person has at most one entry in the table.

- Is this data tidy?



*nor do they have hearts, apparently.

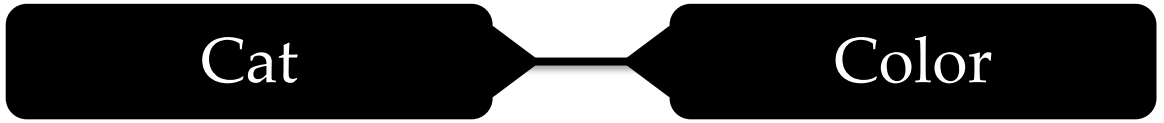
Many-to-Many

- Say we want to keep track of people's cats' colorings:

ID	Name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin
4	Chairman Meow
5	Anderson Pooper
6	Gigabyte

Cat ID	Color ID	Amount
1	1	50
1	2	50
2	2	20
2	4	40
2	5	40
3	1	100

- One column per color, too many columns, too many nulls
- Each cat can have many colors, and each color many cats



Associative tables

Cats

ID	Name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin
4	Chairman Meow
5	Anderson Pooper
6	Gigabyte

Cat ID	Color ID	Amount
1	1	50
1	2	50
2	2	20
2	4	40
2	5	40
3	1	100

Colors

ID	Name
1	Black
2	Brown
3	White
4	Orange
5	Neon Green
6	Invisible

- Primary key ???????????
- [Cat ID, Color ID] (+ [Color ID, Cat ID], case-dependent)
- Foreign key(s) ???????????
- Cat ID and Color ID

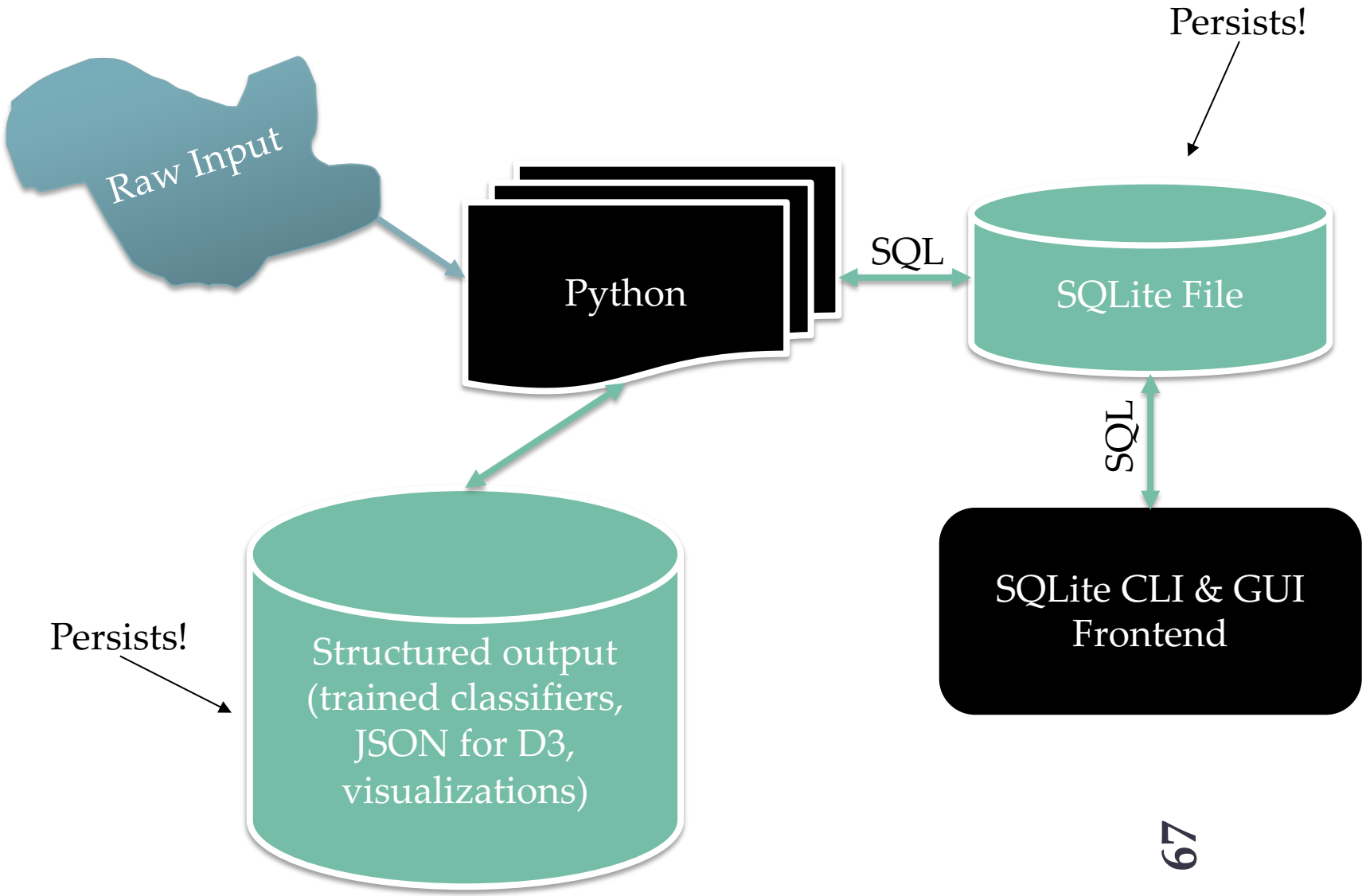
Aside: Pandas

- So, this kinda feels like pandas …
- And pandas kinda feels like a relational data system …
- Pandas is not strictly a relational data system:
- No notion of primary / foreign keys
- It does have indexes (and multi-column indexes):
- pandas.Index: ordered, sliceable set storing axis labels
- pandas.MultiIndex: hierarchical index
- Rule of thumb: do heavy, rough lifting at the relational DB level, then fine-grained slicing and dicing and viz with pandas

SQLite

- On-disk relational database management system (RDMS)
- Applications connect directly to a file
- Most RDMSs have applications connect to a server:
- Advantages include greater concurrency, less restrictive locking
- Disadvantages include, for this class, setup time ☺
- Installation:
- `conda install -c anaconda sqlite`
- (Should come preinstalled, I think?)
- All interactions use Structured Query Language (SQL)

How a relational DB fits into your workflow



Crash Course in SQL (in python)

```
import sqlite3

# Create a database and connect to it
conn = sqlite3.connect("cmsc320.db")
cursor = conn.cursor()

# do cool stuff
conn.close()
```

- Cursor: temporary work area in system memory for manipulating SQL statements and return values
- If you do not close the connection (`conn.close()`), any outstanding transaction is rolled back
- (More on this in a bit.)

Crash Course in SQL (in python)

```
# Make a table
cursor.execute("""
CREATE TABLE cats (
    id INTEGER PRIMARY KEY,
    name TEXT
)""")
```

?????????

id	name

cats

- Capitalization doesn't matter for SQL reserved words
- SELECT = select = SeLeCt
- Rule of thumb: capitalize keywords for readability

Crash Course in SQL (in python)

```
# Insert into the table
cursor.execute("INSERT INTO cats VALUE (1, 'Megabyte')")
cursor.execute("INSERT INTO cats VALUE (2, 'Meowly Cyrus')")
cursor.execute("INSERT INTO cats VALUE (3, 'Fuzz Aldrin')")
conn.commit()
```

id	name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin

```
# Delete row(s) from the table
cursor.execute("DELETE FROM cats WHERE id == 2");
conn.commit()
```

id	name
1	Megabyte
3	Fuzz Aldrin

Crash Course in SQL (in python)

```
# Read all rows from a table
for row in cursor.execute("SELECT * FROM cats"):
    print(row)
```

```
# Read all rows into pandas DataFrame
pd.read_sql_query("SELECT * FROM cats", conn, index_col="id")
```

id	name
1	Megabyte
3	Fuzz Aldrin

- index_col="id": treat column with label "id" as an index
- index_col=1: treat column #1 (i.e., "name") as an index
- (Can also do multi-indexing.)

Joining data

- A join operation merges two or more tables into a single relation. Different ways of doing this:
- Inner
- Left
- Right
- Full Outer

- Join operations are done on columns that explicitly link the tables together

Inner Joins

id	name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin
4	Chairman Meow
5	Anderson Pooper
6	Gigabyte

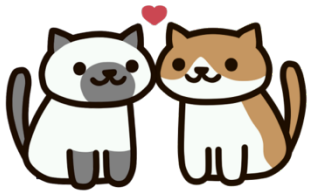
`cats`

cat_id	last_visit
1	02-16-2017
2	02-14-2017
5	02-03-2017

`visits`

- Inner join returns merged rows that share the same value in the column they are being joined on (`id` and `cat_id`).

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
5	Anderson Pooper	02-03-2017



Inner Joins

```
# Inner join in pandas
df_cats = pd.read_sql_query("SELECT * from cats", conn)
df_visits = pd.read_sql_query("SELECT * from visits", conn)
df_cats.merge(df_visits, how = "inner",
              left_on = "id", right_on = "cat_id")
```

```
# Inner join in SQL / SQLite via Python
cursor.execute("""
                SELECT
                    *
                FROM
                    cats, visits
                WHERE
                    cats.id == visits.cat_id
                """)
```

Left Joins

- Inner joins are the most common type of joins (get results that appear in both tables)
- Left joins: all the results from the left table, only some matching results from the right table
- Left join (`cats`, `visits`) on (`id`, `cat_id`) ???????????

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
3	Fuzz Aldrin	NULL
4	Chairman Meow	NULL
5	Anderson Pooper	02-03-2017
6	Gigabyte	NULL

Right Joins

- Take a guess!
- Right join
  (`cats`, `visits`)
  on
  (`id`, `cat_id`)
  ??????????

id	name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin
4	Chairman Meow
5	Anderson Pooper
6	Gigabyte

`cats`

cat_id	last_visit
1	02-16-2017
2	02-14-2017
5	02-03-2017
7	02-19-2017
12	02-21-2017

`visits`

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
5	Anderson Pooper	02-03-2017
7	NULL	02-19-2017
12	NULL	02-21-2017

Left/Right Joins

```
# Left join in pandas
df_cats.merge(df_visits, how = “left”,
              left_on = “id”, right_on = ”cat_id”)
```

```
# Left join in SQL / SQLite via Python
cursor.execute(“SELECT * FROM cats LEFT JOIN visits ON
                    cats.id == visits.cat_id”)
```

```
# Right join in pandas
df_cats.merge(df_visits, how = “right”,
              left_on = “id”, right_on = ”cat_id”)
```

```
# Right join in SQL / SQLite via Python
☹
```

Full Outer Join

- Combines the left and the right join ??????????

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
3	Fuzz Aldrin	NULL
4	Chairman Meow	NULL
5	Anderson Pooper	02-03-2017
6	Gigabyte	NULL
7	NULL	02-19-2017
12	NULL	02-21-2017

```
# Outer join in pandas
df_cats.merge(df_visits, how = “outer”,
              left_on = “id”, right_on = ”cat_id”)
```

Google Image Search One Slide SQL Join Visual

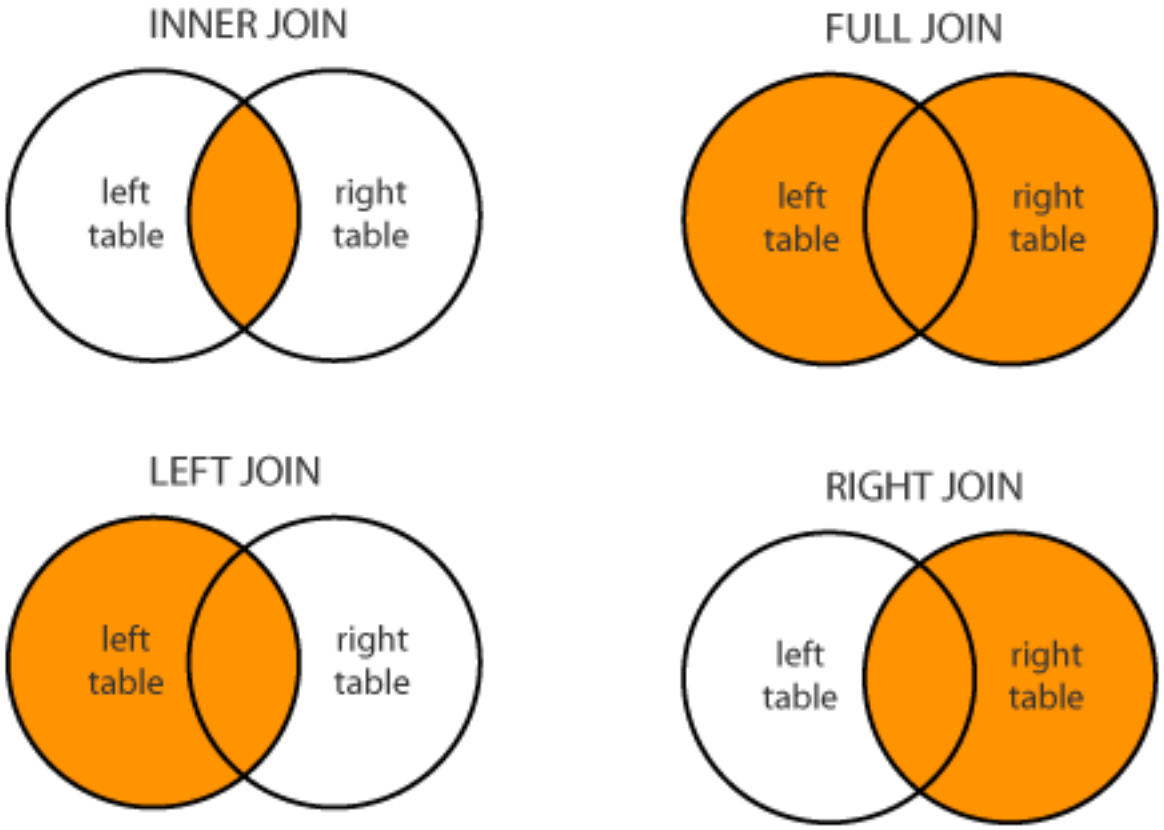


Image credit: http://www.dofactory.com/sql/join

Group by Aggregates

```
SELECT nat_id, AVG(age) as average_age
FROM persons GROUP BY nat_id
```

ID	age	wgt_kg	hgt_cm	nat_id
1	12.2	42.3	145.1	1
2	11.0	40.8	143.8	1
3	15.6	65.3	165.3	2
4	35.1	84.2	185.8	1
5	18.1	62.2	176.2	3
6	19.6	82.1	180.1	1

nat_id	average_age
1	19.48
2	15.6
3	18.1

Raw SQL in Pandas

- If you "think in SQL" already, you'll be fine with pandas:
- `conda install -c anaconda pandasql`
- Info: http://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html

```
# Write the query text
q = """
    SELECT
        *
    FROM
        cats
    LIMIT 10;"""

# Store in a DataFrame
df = sqldf(q, locals())
```