CMSC 320
Introduction to Data Science

Akilesh Praveen
Prof. John Dickerson • Fall 2020 • University of Maryland
https://cmsc320.github.io

Last Revision: May 9, 2020

Table of Contents

1 Notes & Preface **4**

2 Lecture 1 **4**
What is Data Science? . . . 4
Topics . . . 4
Tools . . . 4
Conda . . . 5

3 Lecture 2 **5**
Literate Programming . . . 5
Jupyter Notebook + Alternatives . . . 5
List Comprehensions in Python . . . 5
Using Python3 . . . 6
Python vs. R for Data Scientists . . . 6
The Classic Statistical View of Data . . . 7
Nominal Data . . . 7
Ordinal Data . . . 7
Interval and Ratio Data . . . 8
Data Science at a Glance . . . 9

4 Lecture 3 **10**
Acquiring Data . . . 10
RESTful APIs . . . 10
Oauth . . . 10
GET Requests . . . 11
More on Data Storage Formats . . . 11
SAX . . . 11
Parsing HTML . . . 11

5 Lecture 4 **12**
Available Technologies 12
NumPy Stack 12
Misc About NumPy 12
Linear Algebra with NumPy 13
SciPy 13
The Idea of Reproducibility 13
Best Practices 13
The Idea of Open Data 13
General Process 14
Project Organization 14
A Little on Bias, Ethics, Responsibility 15

6 Lecture 5 & 6 **15**
Tables 15
Selecting / Slicing 16
Aggregating / Reducing 16
Map 16
Group By 17
Group By + Aggregate 18
Union, Intersection, Difference 18
Merge or Join 18
Summary 18
Pandas 18
Dataframes 19
Series 19
Creating a Dataframe 20
Tidy Data 20
SQL and Relational Databases 20
Indexing 21
Relationships 22
SQL and SQLite 23
Joining Data 23
Types of Joins 24
Syntax in PANDAS 26
Visual Example 27

7 Lecture 7 **27**
Centralized VCS 27
Distributed VCS 28
Branching 28
When Should I Branch? 28
Git Basics 28
Online Hosting 29

8 Lecture 8 **29**
Missing Data 29
Ol' Reliable 30

9 Footnotes 30

1 Notes & Preface

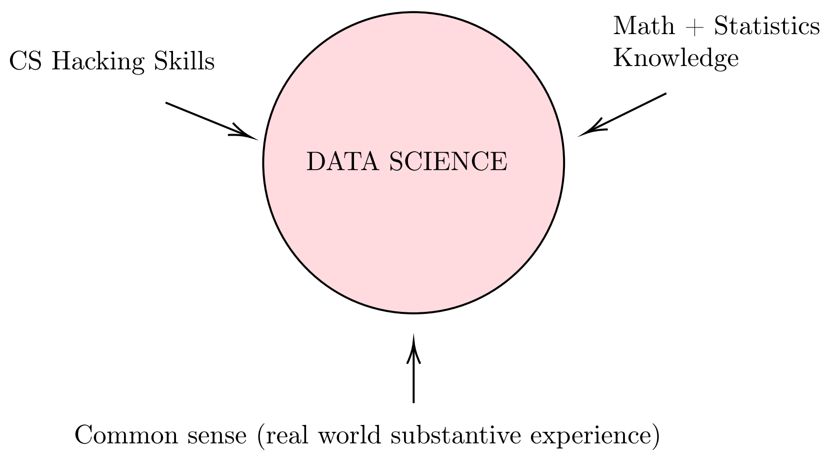
Course notes for CMSC320, under Prof. John Dickerson. Notes collected from previous and current lectures.

2 Lecture 1

What is Data Science?

Data Science is the application of computation and statistical techniques to address or gain insight. It's the intersection of statistics and Computer Science. Based on what I've learned thus far, learning to do data science is like learning how to use a TI-84 in statistics class. You're simply learning how to leverage programming tools in order to perform advanced, complex, and meaningful data-related operations.

It's the use of statistics and computer science in order to find real-world insights.



Topics

Here are the general topics that this class will cover.

- Processing data
- Visualizing data
- Understanding data
- Communicating data
- Extracting value from data

Tools

Here are some tools commonly employed by data scientists. We'll try to cover how to use most of them here.

- Python
- Scikit-Learn

- Docker
- PANDAS
- Spark
- TensorFlow

Conda

Conda is a package and environment manager for python that we can use with the command line. We can create multiple environments for us and install separate packages in each of them. This will be highly useful to us, as we sometimes want to consolidate the tools we use into separate environments.

3 Lecture 2

Definition:

Data Collection → The process of measuring and gathering information on targeted variables.

Literate Programming

The idea of **literate programming** is that you have the source code, an explanation of the source code, and the end result of running the code all in one file. Usually, this file is identified as a *notebook*. In other words, the syntax is no different from regular code, you just get a more organized way to show off tables, plots, and other outputs generated from your code.

Jupyter Notebook + Alternatives

Jupyter Notebook is a service that started off as `iPython`, but it's basically a web-based platform that we use for literate programming. Specifically, it supports Python-based literate programming. Most data scientists prefer it, and it can also apparently leverage big data tools, such as Apache Spark.

It saves files in `.ipynb` format, which most platforms (i.e. GitHub) have built in viewers for. Options to export in other readable formats are available. Basically, it's just Python with a bunch of bells and whistles on top to make the output of your code look pretty.

Apache Zeppelin is an alternative data analysis tool, but we will stick to Jupyter for our purposes. This is because Jupyter seems to be preferred in industry.

RStudio is the equivalent, for people who prefer to use the `R` programming language for data science.

This course will be centered around Jupyter Notebook.

List Comprehensions in Python

To make lists in Python, you can use loops or the `map()` function, but a *pythonic* way of doing this would be to use a list comprehension. Below is a simple example.

Example: Make a list of all the squares of `{0,1,2,3,4,5,6,7,8,9}`

List Comprehension:
```
squares = [i * i for i in range(10)]
```

A good way of thinking about this is that it allows you to build sets like a mathematician. This is a common theme in data science, where we can find the intersection between a lot of math stuff and computer science stuff. It's good to know how lists are generated in a mathematical sense in Python for that reason. Here's an example where we translate mathematical notation into a Python list comprehension.

Example: Make a list of all odd natural numbers from 0 to 999

Math Notation:
$E = \{x \mid x \text{ in } \mathbb{N} \wedge x \text{ is odd } \wedge x < 1000\ \}$

List Comprehension:
```
E = [x for x in range(1000) if x % 2 != 0]
```

Using Python3

We will use Python3. Since I used Python2 during my internship, I'm going to note some big changes to keep track of.

- Python3 is backwards incompatible. (Don't write in Python2!)
- Print has changed from a command to a function, so make sure to use proper function notation when invoking it.
- Division has changed. `1/2` no longer equals 0. `1/2 == 0.5` and floored division is now taken care of this way: `1//2 == 0`

Python vs. R for Data Scientists

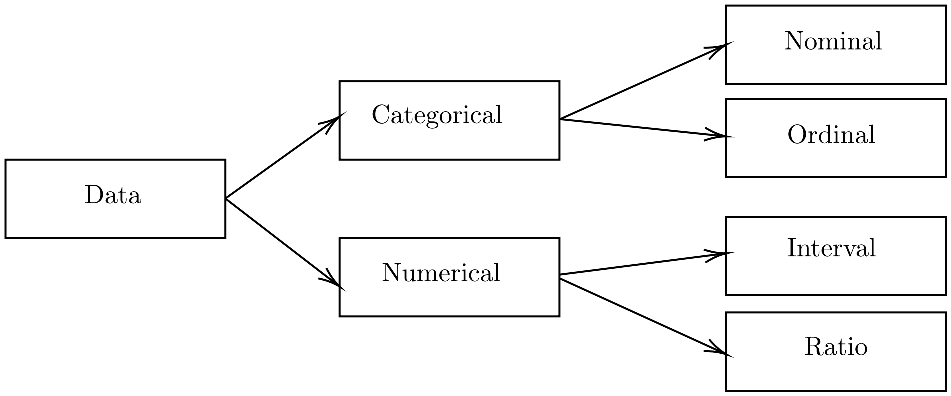
Some arguments for both sides in terms of what to use.

- Python is a 'full' programming langauge. Also, if you've got prior experience with Python paradigms or just programming in general, that's a big plus in terms of learning curve.
- R has more mature 'pure statistics' libraries, but Python is apparently catching up.
- In terms of **processing speed**, R is certainly faster. It was designed and optimized for statistics processing.
- Python is preferable for machine learning operations, which is pretty big right now.

My personal choice will be to use Python as much as I can when I'm studying this course. Since it's more prominent in the tech industry, I should be using it more anyway.

The Classic Statistical View of Data

There are **four** main types of data: Nominal, Ordinal, Interval, and Ratio data. They can each be classified under two main subgroups, Categorical and Numerical data. Here's a visualization.



Nominal Data

A type of categorical data, nominal data value have names and describe the state of things. For example, your marriage status is nominal data because you can either be *single*, *married*, or *separated*. Another example is the type of drink you're going to have. Will it be *Milk*, *Beer*, or *Juice*?

The key here is that there can be no quantitative values assigned to each of these categories, as that would allow us to do math with them and would defeat the purpose of these labels. These values **cannot be easily compared**, so they have no material value. *E.g. being single is not quantitatively better than being married (objectively), and vice versa.*

> **Example:** What is your marital status?
>
> - Married
> - Divorced (separated)
> - Single

Ordinal Data

Ordinal data represents values that have names that describe the state of things, but in this case, there **is** an ordering of those values. This is what sets it apart from nominal data.

> **Example:** What did you think of the movie?
>
> - Strongly liked
> - Liked
> - Indifferent
> - Disliked
> - Strongly Disliked

Given how subjective some of these things can be, the distinction between nominal and ordinal can be **blurry** at times. For example, going back to our nominal example, some people may think that being single is quantitatively better than being married.

Interval and Ratio Data

Interval and Ratio data are pretty similar, and both can be used to measure things that can be represented by either integers or real numbers.

Interval data scales with fixed but arbitrary values. That might sound silly, but a good example is **dates**. Below is an example of two data comparisons of interval data that seem arbitrary, but indeed hold integer value.

> **Example:** The following two operations are equal.
>
> ```
> 10/1/2019 - 9/1/2019
> 10/1/2018 - 9/1/2018
> ```

The measures don't look like integer values at first, but we can quantify them by marking them with days.

Here's what sets **Interval** data apart, however. You have **no method** of computing ratios or scales with it. For example, never mind that you can try computing `(9/1/2019` $\times$ `8/25/2015)`, the unit of the answer would be totally useless to us, and neither would the actual number, even if you went ahead with the operation.

Ratio data is, in essence, the same as interval data in that it is numerical, but the scale itself **has a true zero**. While dates don't necessarily have a true zero, we can say that money counts as ratio data. For example, having zero money means that you're at the absolute zero of that scale, whereas the absolute zero for dates is disputable. Are we saying we're starting at O A.D.? The Big Bang? Even earlier?

Differentiating between the two is usually a case-by-case basis thing, which is what I'm thinking is the best way to handle any conflicts I end up running into between ratio and interval data.

> **Example:** Interval data
>
> Temperature on the scale of Celsius or Fahrenheit is interval-type data because $0°$ is set to an arbitrarily fixed point. Also, we can't scale it properly- $30°F$ isn't twice as hot as $15°F$.
>
> **Example:** Ratio data
>
> Temperature on the Kelvin scale is ratio data. $0K$ is set at legitimate absolute zero, and $50K$ is truly twice as cold as $100K$.

Data Science at a Glance

Data science is basically manipulating and computing using data. As such, we need to shift our thinking from writing **imperative** code to manipulate **data structures** to creating **sequences and pipelines** to conduct operations on **data**. That stuff is covered more in 420 and 424, for reference.

More often than not, we have to take the data that we've found and make it easily understandable for humans. This is called Data Representation.

> **Definition:**
>
> **Data Representation** → The natural way to think about data, in a human way.

Here are some ways that we think about data in this class:

- **One Dimensional Arrays** → E.g. `<'red', 'blue', 'green'>` or `<0,3,4>`. We can use functions like map, fold and filter to manipulate these.
- **N-Dimensional Arrays** → Also known as **tensors**.
  - For example, a Tensor of dimensions `[6,4]` is just a 6×4 matrix.
  - Similarly, a Tensor of dimensions `[4,4,2]` is a 3D array.
  - Here, we can start to make use of **Linear Algebra** for further data manipulation. Some example operations that we can use to mess with tensors:
    - Matrix/Tensor Multiplications
    - Transpose
    - Vector Multiplication
    - Matrix Factorization (we will explore this later)
- **Sets** of objects, or **Key/Value Pairs**
- **Tables/Relations** → This goes into relational databases, which is the basis of `SQL`. We'll go into this later.
- **Hierarchies/Trees/Graphs** → This sort of spills over into data structures, but they've got some additional nuances included with them.
  - They tend to make use of 'path' queries
  - Graph and Tree Algorithms will be useful here, efficiency is key
  - Example: networks are represented this way, we'll cover that later in this class

4 Lecture 3

Acquiring Data

Here are some examples of how we can grab data from places. Pretty obvious, common sense stuff. We're going to explore all of these as we move forward.

- Direct download from online or loading it from local storage
- Generate the data locally via a simulation or equivalent program
- Query data from a database
- Query data from an API
- Scrape data from a website

When you pull from APIs, you're going to want to be using HTTP Requests.

RESTful APIs

This stands for REpresentational State Transfer APIs, and it's basically a standard that enforces that APIs do a few things. It says that they should support these basic operations:

- `GET` → Query a data entry
- `POST` → Create a new data entry
- `PUT` → Update an existing data entry
- `DELETE` → Delete an existing data entry

RESTful APIs are also supposed to be stateless. That is, with every API request, you send a token of who you are, and you get a current capture of the data at that time/edit the data.

A good example of a REST API is Github, where you can use REST API calls on your repositories.

There are other guiding principles and miscellaneous guidelines for RESTful APIs, which can all be found at `https:/restfulapi.net`

Aside: GRAPHQL

GraphQL → REST has been adopted by many developers and is widely regarded as the traditional way to send data over HTTP. GraphQL, on the other hand, is a revolutionary new player that's presented as a way to *replace* legacy REST APIs *(back4apps)*

Oauth

If you want to grant an app access to your identity without actually giving it your username and password, is there a way to do that? The answer is **yes**, because this is a common software engineering problem.

OAuth is the standard for *access delegation* used for internet users to grant websites access to their information on other sites. A pretty good example of this is Google's sign in page on other websites. How do you think other websites conduct sign in without knowing your password for your Google account?

GET Requests

Assume we used Python's `requests` module to query a server with a `GET` request.

First, we'd get either a `CSV`, `JSON`, or `HTML/XML/XHTML` file back, in response. This is the data that we have to sift through. *Note:* You might also get a domain-specific file, like an **rvt** file. You're always welcome to make your own filetype for storing data, but make sure it's actually documented somewhere.

Aside: Parsing CSVs and JSON

Never write your own `CSV` or `JSON` parsers. This is another example of reinventing the wheel. We'll use Python Libraries to do this more easily. *E.g. PANDAS*

More on Data Storage Formats

Definition:

Serialization → The process of converting objects into strings.
Deserialization → The process of converting strings back into objects.

`JSON` is a pretty common format for serializing objects. Plus, serializing objects makes it easier for humans to read and perform sanity-checks on. In Python, `JSON` is built with Strings, Lists, Dictionaries, and sometimes mixes of a few of those together.

Definition:

Document Object Model → A tree-based data storage method. For example, HTML is structured this way.

SAX

SAX is a lightweight way to process XML. It generates a stream of events as it parses an XML file. IT helps us pay attention to individual parts of an XML file without having to process through the rest of it.

Parsing HTML

Parsing HTML is the hardest to do in this case, as I've seen many times before in hackathon projects. Although HTML's specifications are pure, the real world examples of it are pretty nightmarish, thanks to how it interacts with JavaScript and loads dynamic content. All in all, it's fairly unreliable in terms of parsing it manually.

In this case, we're best off using the Python library `BeautifulSoup`. We can also make use of Python's Regex, which is similar enough to Ruby regex that we worked with in 330. A website like Rubular-`https://pythex.org` will be useful in this case.

By combining `BeautifulSoup`, Regular expressions, and `GET` requests, we can make the process fairly streamlined. This is usually what we'll be using to scrape websites. In order to scrape more dynamic websites, we'd probably have to make use of Selenium. Check my 320 folder to find an example of a simple webscraper with `BeautifulSoup`.

5 Lecture 4

Overview: Numpy, PANDAS, Relational Databases, Apache Spark

Available Technologies

Python has a bunch of 3rd party packages for scientific and numerical computation. Some examples are..

- **`Numpy and Scipy`** → Numerical and scientific function libraries.
- **`NUMBA`** → A Python compiler supporting 'Just in Time' compilation. That is, it supports compilation of code while code is running.
- **`ALGLIB`** → A cross-platform numerical analysis library
- **`PANDAS`** → An extensive data analysis tool with some neat built-in data structures
- **`PyGSL`** → GNU Scientific Library in Python
- **Scientific Python** → A collection of scientific computing modules for Python

These are a bunch of examples of what's available to developers right now, but we won't focus on all of it. Particular emphasis will be placed on `Numpy` and `PANDAS`.

NumPy Stack

The **NumPy** stack is the most commonly used out of all of these packages. It includes the following:

- `Numpy` - Works sort of like MatLab, just lets us handle a lot of number manipulation and mathematical operations
- `Matplotlib` - This is a plotting and graphing library
- `PANDAS` - This gives us a bunch of data structures and data analysis tools to play with/keep track of our data. (Usually, you'll want to import your data into a `PANDAS dataframe` or something.)
- `SciPy`
- `SymPy`
- `Jupyter` - This will be our medium for **literate programming**.

To see more about this stuff, search Google for the **NumPy Stack** and you'll find everything you need.

Misc About NumPy

Here are a few more notable things about `Numpy`:

- It contains the **n-dimensional array** object
- It contains 'sophisticated' functions that we can use
- It provides us with excellent tools to integrate `C++`, `C`, and even `FORTRAN`
- It has math capabilities that are highly useful to us (e.g. Linear Algebra, Fourier Transform, etc)
- `Numpy` also comes with a bunch of new DataTypes for us to use.

Aside: Numpy Arrays

Arrays in `Numpy` are different from regular lists in Python, so make sure your syntax is correct and you know the difference when you decide to use either one in practice.

Linear Algebra with NumPy

One of `NumPy`'s most common uses lies within its **Linear Algebra** module. It allows us to do regular LA stuff, like `.transpose()` and `.inverse()` to matrices stored as n-dim arrays. Here's an example.

```
# Note: remember, we have to use NumPy's n-dimensional array object here

array([[1.0, 2.0],
       [3.0, 4.0]]).transpose()
```

SciPy

`SciPy` includes various tools and functions for solving common problems in **scientific computing**.

We won't use it much for now, but it's supposed to be good to know. Often you'll be able to find higher-level `Scipy` functions that will work around the need to call lower-level `Scipy` functions. It's got a lot of functionality built in, so make sure not to overlook it.

The Idea of Reproducibility

Starting from the same dataset, can we reproduce your analysis and get the same results? **This is the goal that we're trying to fulfill with our analysis-** we want our stuff to be reproducible! (Otherwise, what exactly does it even mean?)

Best Practices

Honestly, most of this stuff should be common sense.

- Use version control to keep track of code. (e.g. `git`)
- Use unit testing. (e.g. `unittest` module in python)
- Use libraries when you can. (don't reinvent the wheel!)

The Idea of Open Data

Some data should be widely available for everyone to use as they want, without restrictions from copyright, etc.

This is probably where all of our free data comes from, so this idea is super helpful to us as data scientists.

General Process

Here's the general process for data science- just so we have an idea of what's going on.

Data Acquisition

Acquire the data, then put it in a usable format.

→

Algorithm / Tool Development

If new tools are required to analyze your data, create them.

If new algorithms may be needed to analyze it, synthesize them.

→

Data Acquisition

Use the tools you've built / setup and the algorithms you've created or implemented to analyze the data.

After we do that, we still technically have some programming left. In this new era of literate programming, there's one more step of processing we have to do with our results in order to make them publicly presentable and meaningful.

Communication of Results

Use the principles of **literate programming** to provide results, plots, and publication regarding your data science experiment. This gives you the ability to explain your choices and clearly interpret your results.

It's emphasized a lot here to think like an **algorithm developer**, as you're going to need efficiency in the data analysis that you perform. However, you also need to think like an experiment-conducting **data scientist**. We don't usually get enough training as the latter, so hopefully this course should be an introduction to that sort of stuff.

Project Organization

Make sure to organize your project in folders appropriately. Specifically, even if you have a lot of components, group with with a focus on experimental procedure.

You should certainly be isolating things like **data**, **tools**, and **experiments** into their own folders. Data could include your raw input data, along with data that you've done some processing on. Tools could include Python environment you're using, and experiments could include the meat of what your data science work will be- pipeline scripts, results, figures, plots, analysis scripts, etc.

A Little on Bias, Ethics, Responsibility

Aside: Fairness Through Blindness

The concept of not letting an algorithm look at protected attributes in order to keep it from forming potentially harmful biases.

A great example of fairness through blindness could be software that determines the outcome for a loan application. We want the results to be **independent** of an applicant's race, but they can be **dependent** on non-protected attributes, such as credit history and income.

Aside: FATML

FATML stands for Fairness, Accountability, and Transparency in Machine Learning.

Overall, here are some guiding principles for data ethics:

- Start with clear user need, with a focus on public benefit. (Can't go wrong with this!)
- Use data and tools that emphasize **minimum** intrusion/invasion of privacy. (Sometimes, we have no choice but to handle sensitive data)
- Create robust data science models that minimize bias and focus on objective accuracy.
- Be alert to public perceptions.
- Be open and accountable for your actions.
- Security is key- especially if working with sensitive data.

6 Lecture 5 & 6

Big ideas: Pandas + Relational Databases

- Tables (Specifically, the abstraction + operations)
- `PANDAS`
- The idea of 'tidy' data
- `SQL`

Tables

Here's the idea- we can abstract data into our own little data structures just like computer scientists do, and a lot of the time, in data science, tables are the optimal way to do that. (This is why software like PANDAS and Numpy have excellent support for these structures.)

Variables (Attributes/Labels/Columns)

id	age	weight	height
1	12	42.3	145.6
2	11	50.9	161.2
3	13.5	61.3	181.3
4	14	41.4	121.4
5	15	55	135.5

Observations (rows, tuples)

Known as the Index (ID) column. Usually, there are no duplicates of these allowed, and they're often ordered like this.

Here's an example table. I've highlighted and color coded the important aspects of it. Remember, don't think of this as the data structure itself- this is just an abstraction to help us keep track of our vast amounts of data. However, most table implementations do a pretty good job of representing the stuff I've color coded.

Selecting / Slicing

Selecting one or more of the rows or columns in particular to analyze. Examples:

- Select only columns ID + Age
- Select all rows with weight > 41
- We can also apply a combination of the above 2. (*You can combine select rules!*)

Aggregating / Reducing

Combining values across a column into a single value. (We don't do this across rows, because that obviously wouldn't make any sense. Think about it!). Examples:

- Find the sum of all row's columns
- Find the max of the weight column

Note: It's usually never useful to aggregate/reduce the ID column, so for most cases, we ignore it when we perform such operations.

Map

Apply a function to every row, possibly creating fewer or more columns. This one's a little weird to think about without a clear example, so I'm including one below.

map

id	address
1	College Park, MD 20742
2	D.C., D.C. 2000
3	Cupertino, CA 95014

id	city	state	zip
1	College Park	MD	20742
2	D.C.	D.C.	2000
3	Cupertino	CA	95014

map

Notice how applying map to either table is valid in this case- sometimes we want to break down columns into more specific values, and sometimes we want to combine them into singular columns. Each of these operations is totally valid, and has its uses. (This is evident in the projects for this class).

Again, this is mostly about what you need. There's no necessary better or worse in this case (more columns does not always equal better data).

Group By

Group By is an operation that allows you to group tuples together based on the values in columns/dimensions. Let's say we had the following table of house addresses like earlier. This time, we'll add the number of people in each house as a column as well.

id	city	state	zip	people
1	College Park	MD	20742	3
2	Washington	D.C.	2000	4
3	Cupertino	CA	95014	3
4	College Park	MD	20742	2

Let's say we only wanted to see the data from a **single city**. In this case, let's pick College Park.

id	city	state	zip	people
1	College Park	MD	20742	3
2	Washington	D.C.	2000	4
3	Cupertino	CA	95014	3
4	College Park	MD	20742	2

This is what a 'Group By' operation would be perfect for. It'll basically just get us the rows that are from the city that we want.

Group By + Aggregate

We can combined Group By and Aggregate in pretty cool ways to get results that we want. For example, let's say we wanted to leverage the above table and get the sum total of all people who live in College Park, D.C., and Cupertino, respectively. By using a combo of Group By and Aggregate, we can totally do that. (*Group By* City, then perform summing *aggregation* operation.)

Union, Intersection, Difference

These are your usual set operations from statistics. However, this only works if the tables have identical attributes (columns). If they have identical columns, they are called **compatible tables**.

Examples: (Table A) $\cup$ (Table B) results in (Table C) where all three tables have the same attributes. Likewise, (Table A) $\cap$ (Table B) results in (Table D) where all three also have the same attributes.

Merge or Join

This is how you combine rows across tables, based on some distinguishing element (i.e. ID column). For example, you'd basically take the row with ID 1 in your first table and add all those elements to the row with ID 1 in your second table.

There isn't a graphical example here just because we'll be talking about this a lot more in depth in later lectures. For now, just remember it as a way to combine tables.

Summary

Overall, **Tables** are a simple and common abstraction. They're how we mainly keep track of data when we do most of these data science things, so it's worth learning how to employ them, and what basic operations we can use when manipulating them.

These **operations**, at a glance:

- Select
- Map
- Aggregate/Reduce
- Join/Merge
- Union/Concat
- Group By

Keep in mind that tables are an *abstraction*, after all, so these operations may be named different things in the languages you use to manipulate them. That's why its important to not just memorize the names of these operations, but what they actually do. This will prepare you for work with any data table manipulation program.

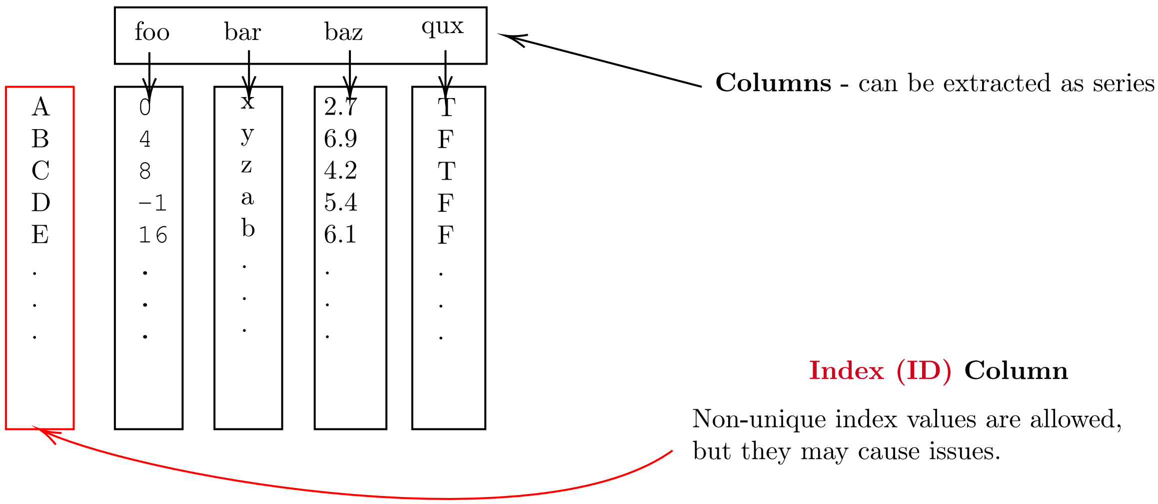
Pandas

`PANDAS` is a data manipulation library for Python that's highly optimized for performance. It contains two key constructs, **`Dataframes`** and **`Series`**.

Dataframes

This is PANDAS's way of representing the table abstraction we were looking at. Geeksforgeeks even calls this one a tabular data structure.

There are a lot of PANDAS-specific commands that you'll have to learn to be proficient with these, but Geeksforgeeks is a great reference for them. (You'll find it's pretty easy to conduct all the basic table operations)



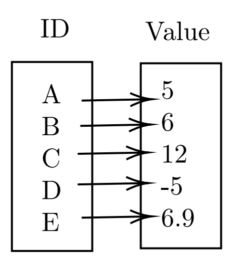
Series

Think of a `series` as a subclass of Numpy's `ndarray`.

Geeksforgeeks calls this a 'one dimensional labeled array capable of holding any data type'. Think of it as a column in an Excel spreadsheet. In fact, their most common use is when you pull a single column out of a `dataframe` and want to analyze it individually.

The object itself supports integer and label based indexing (like letters), and allows us to perform a bunch of operations involving the index.

To create one of these we can grab a column from a `dataframe`, or we can create one out of a regular Python `list` or a Numpy `ndarray`.



For this sort of stuff, you can probably look at **GeeksforGeeks** for more information. They have excellent documentation on PANDAS functionality.

Creating a Dataframe

To **create a `dataframe`**, you have a variety of options.

- Get data directly from a Python dictionary, a bunch of series, or other data structures
- `Pandas.read_csv()` → take in data from a `.csv` file
- `Pandas.read_excel()` → take in data from an excel spreadsheet
- `Pandas.read_html()` → take in data from a (static) website (e.g. a website with a big table of data on it)
- From a database by using SQL to make queries
- From clipboard, URL, many more options.

Tidy Data

There are **3** components of **tidy data**: **Labels**, **Variables** (values), and **Observations**.

Variables

Age	Weight	Height
12.2	42.3	145.1
11.0	40.8	143.8
15.6	65.3	165.3
35.1	84.2	185.5

Labels (name of the whole row)

Obvservations

Here's some elaboration.

- **Variables** → A measure or attribute, e.g. age, weight, height, sex.
- **Value** → Measurement of a *singular* attribute, e.g. `12.2 lbs` or `5.9 inches`.
- **Observation** → All measurements for an object; a *row* in the table. E.g. a single observation in the above table would be `[12.2, 42.3, 145.1]`.

Tidying and melting data basically just means that you mix data around until it's nice and usable. Usually, you are tidying in pursuit of a specific use-case, so less columns or more columns are never the 'better' option. This is one of those things that takes practice and application.

TL;DR Clean up and organize your data before you mess with it!

SQL and Relational Databases

Big Question: What is a relation?
Answer: In a databases context, they usually mean, "a tabular set of data either permanently stored in the database (a table) or derived from tables according to a mathematical description (a view or a query result)." (*Larry Lustig, Stackoverflow*)

Definition:

Relation → A relation is a data structure which consists of a heading and an unordered set of tuples which share the same type.
Relation Schema → A list of all attribute names and their domains. E.g. 'The Schema for a Table'.

Indexing

Definition:

Index → An auxiliary data structure of a relation database designed to speed up the retrieval of rows.

How can we leverage **indexes** to improve search times for our relational databases? Take a look at the example below. Let's say we wanted to find all people from Canada (with a nat_id of 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

Unfortunately, the time it takes for us to build this list every time we want to leverage the result of this search is **`O(n)`**. This is not so great for us. However, if we decide to build an **index** on the 'nat_id' attribute, things change.

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

Now, after establishing this index, which acts like a hidden sorted map of references to a specific attribute in a table, we are allowed **`O(log n)`** lookup with the parameter nat_id.

You can choose to build an index on a certain attribute to improve search times for it, but they aren't free. They're expensive- establish an index with caution. Not only do they take time to initially build, but now, whenever you add to the table or update it, you also need to update the index. In that sense, establishing too many indexes could lead to other operations, such as changing table values, taking a very long time. It's a delicate balance.

Aside: Indexes

Indexes are actually implemented with data structures like **B-trees**, which is why we are able to perform data access in `O(log n)` time. The worst case height of a B-tree is `O(log n)`, and since a search is dependent on height, B-tree lookups run in `O(log n)` on average.

Relationships

Primary keys and foreign keys determine interactions between different tables. These are formally known as relationships. First, let's establish definitions for primary and foreign keys.

Definition:

Primary Keys → Columns whose data can be used to uniquely identify each row in the table. Highlighted in red below. (E.g. the ID column)
Foreign Keys → Columns that refer to the primary key in another table. Highlighted in blue below.

Foreign Keys

Primary Keys

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

There are four main types of relationships between keys. Here they are with some examples.

- **One-to-many / Many-to-one**: A person can have one nationality, but a nationality can have many people associated with it.
- **One-to-one**: People each have one unique SSN- no conflicts.
- **One-to-one-or-none**: People can either have 1 car, or no car.
- **Many-to-many**: Cats and colors. Red can be on many cats, and many colors can be on a single cat.

Even though this system can sort of be replicated in `PANDAS`, `PANDAS` is not strictly a relational data system. Notably, it doesn't have notions of primary or foreign keys built in.

> Do heavy, rough lifting at the **relational database** level, (e.g. when you're deciding what sort of `SQL` queries to make) and then do the fine-grained slicing and dicing and visualization with **`PANDAS`**.

SQL and SQLite

Definition:

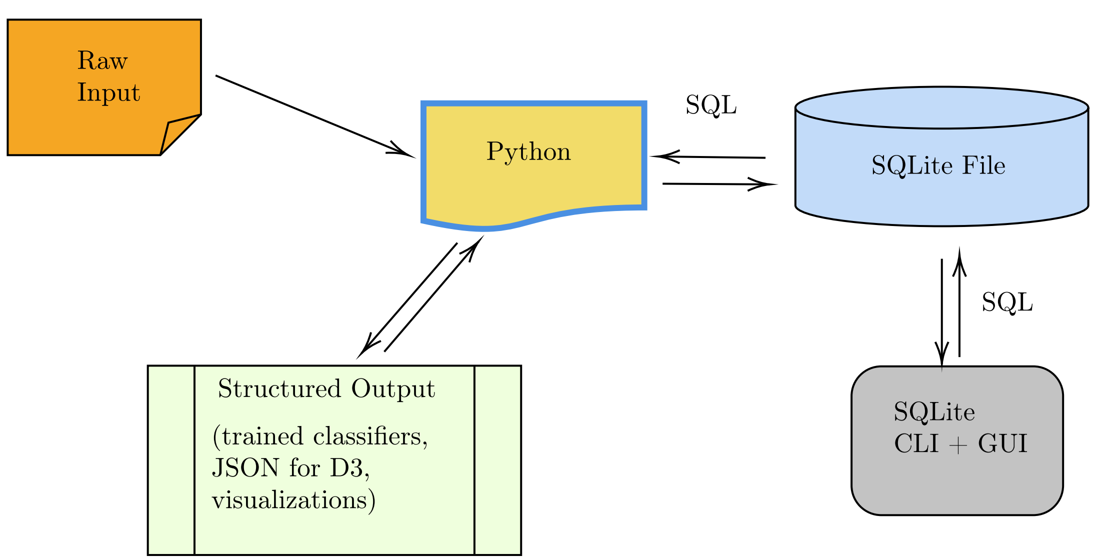
SQL → Stands for 'Structured Query Language', and is the ANSI-standardized way for us to communicate with relational databases. Standard SQL commands like "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database.

We use **SQLite**, an on-disk relational database management system, in order to interact with our databases. Most **RDMS**s connect to a server, which provides support for better concurrency, that sort of stuff takes longer to setup.

On the other hand, `SQLite` is pretty simple to install via `conda`, so we're going to go ahead and use it.

`SQLite` provides us with two main ways to communicate with the `SQL` database that it maintains. First of all, it gives us a cool GUI-based environment where we can deal with manipulating data manually, but it also allows us to write `SQL` statements to interact with it, whether that be from the command line or from within Python.

Here's an example of how a relational database fits into our workflow.



To work with `SQLite` in Python3, simply install and import the `SQLite3` package, and use that.

Joining Data

A **join** operations merges two or more tables into a single relation, based on their columns. There are four total types of join operations.

> Formally, the way we format join statements is the following:
>
> `<type of join> join (<left table>, <right table>) on (<left table column>, <right table column>)`
>
> Example: `Right join (cats, visits) on (id, cat_id)`

Types of Joins

- **Inner Join** → Returns merged rows that share the **same** value in the column that they are being joined on. Let's say we had the following two tables and wanted to join them.

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`

In order to perform an inner join on these two tables, we need to pick a column to 'join them on'. Here, let's inner join these two tables on `id` and `cat_id`. The result would look like this:

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

Inner joins are the most common type of joins; they get us the results that are shared by both tables.

- **Left Join** → A left join gets us **all** the results from the **left** table, but only **some** (the corresponding matching results) from the **right table**. So, what happens if we Left Joined `cats` and `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

You'll notice that the fields that we couldn't fill out get populated with `NULL`.

- **Right Join** → A right join gets us **all** the results from the **right** table, but only **some** (the corresponding matching results) from the **left** table. Here's an example, with updated `cats` and `visits` tables.

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`

If we were to perform a right join on these two tables, here's what would happen. It's basically just a flipped version of the left join.

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

Again, this time, notice how the row entries missing from the left table are now set to `NULL`

- **Full Outer Join** → The full outer join combines the left and right join. It's analogous to a **union** operation. Here's an example of a full outer join of `cats` and `visits` on `id` and `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
7	NULL	02-19-2017
12	NULL	02-21-2017

Syntax in PANDAS

Here's how to write some basic join syntax in `PANDAS`. This should be easy once you learn how to phrase join statements- you're basically just translating it into code.

First, this is how you'd read from `SQLite` (or any other database you're hooked up to) using `Pandas` and generate the appropriate dataframes to work with.

```
# establish dataframes from SQL
df_cats = pd.read_sql_query("SELECT * from cats", conn)
df_visits = pd.read_sql_query("SELECT * from visits", conn)
```

Now, here's how to do the joins.

```
# inner join
df_cats.merge(df_visits, how = "inner",
              left_on = "id", right_on = "cat_id")

# left join
df_cats.merge(df_visits, how = "left",
              left_on = "id", right_on = "cat_id")

# right join
df_cats.merge(df_visits, how = "right",
              left_on = "id", right_on = "cat_id")

# full outer join
df_cats.merge(df_visits, how = "outer",
              left_on = "id", right_on = "cat_id")
```

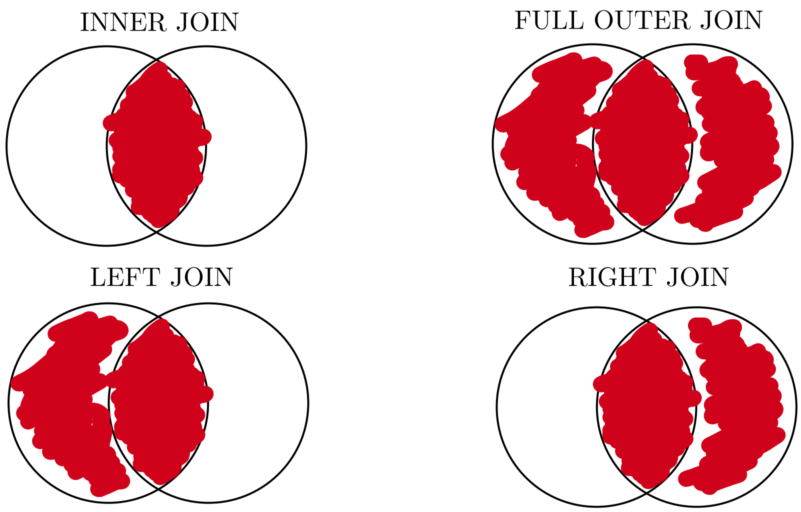
There are also ways you can perform most of these joins via SQL (save for the right join), but I would prefer doing them from Python. As such, I won't include the `SQL` syntax here.

Aside: Raw `SQL` with Pandas

If you want to use raw `SQL` queries to interact with `PANDAS` dataframes, you are free to do so when you use the `PandaSQL` library.

Visual Example

Here's a neat way to visualize joins using Venn Diagrams.



7 Lecture 7

This lecture focuses mainly on **version control** and **`Git`**. Since I know the basics of this stuff already, this will just be a smaller review of `Git` plus the stuff I didn't already know very well coming into this course.

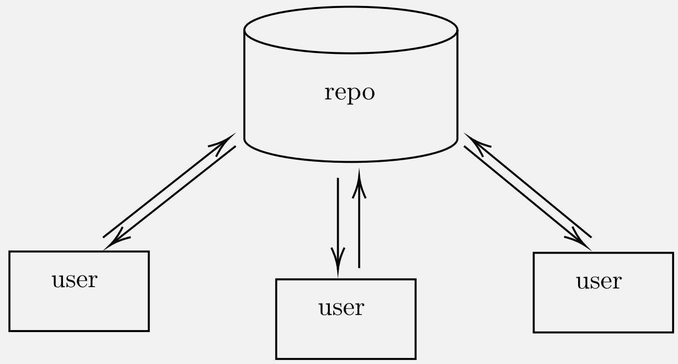
> **Big Idea:** Teams needed good ways to maintain central repositories for their code projects without having conflicting versions of their code, so **version control software** was created.
>
> Eventually, this software carried the secondary purpose of tracking and managing bugs. These days, however, dedicated enterprise tools like **JIRA** also exist to handle bugs. It's mainly based on what your company decides to use.

Version Control Software is used to do the following:

- Search revision history and get **any version** of the project you're working on
- Share code changes with your collaborators
- Confidently make changes to large files

Centralized VCS

People used **Centralized VCS** to have multiple users all pushing towards a central repository. I've seen examples of this in use at HPE (`SVN`- Subversion), and during my classes at UMD- `CVS` in CMSC132.



In this case, there's a singular centralized codebase that users will all be contributing to at the same time.

Distributed VCS

Distributed VCS has no central repository, and every repository has their own commits and history. Examples of this would be **Git** and **Mercurial**.

Aside: `Git`'s popularity

`Git` is currently the most widely used code management tool (VCS). The next two in line are `SVN` and `Mercurial`.

`Git` is also more efficient and secure than `SVN`, but a lot of old legacy codebases and companies still make use of `SVN`. For that reason, it may be good to pick up some of the basics of `SVN` so I'm not totally unfamiliar, but `Git` generally seems to be preferable.

Branching

`Git` also allows for branches. **Merging** also ties in closely with branching on git, and has a lot more sophistication in `Git`.

When Should I Branch?

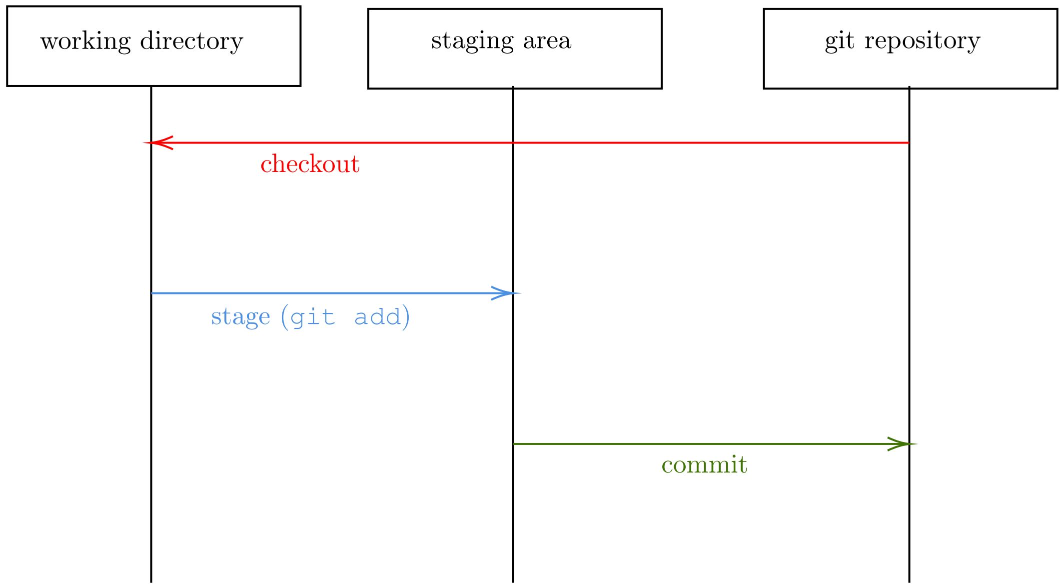
Anything in the master branch is considered to be deployable. If you're adding a new feature, working on an experiment, or trying to implement a new fix, make a branch. You can always merge it back to master once it's considered 'deployable' again.

Git Basics

As far as `git` is concerned, a file has 4 states:

- **Modified** → File has been changed, but those changes have not been committed
- **Staged** → Marked to go to next commit snapshot
- **Committed** → Safely stored in local database as part of a 'commit'
- **Untracked** → News added or removed files

This idea can be complemented by a visual guide, where you can see the three main 'places' that a file can be within the `git` system.



Online Hosting

Github, **Bitbucket**, and **Gitlab** are all popular sources that will host your git repositories. Think of this as just another place for your git stuff to exist, except whenever you want to update the main online repository, perform a `git push`.

It's sort of like a hybrid of a centralized and distributed VCS. According to a website online, "GitHub and similar services bring all of the benefits of a decentralized VCS to a centralized service."

8 Lecture 8

Missing Data

Missing data is information that we want to know, but don't know. It can come in many forms. Here are some examples.

- People omitting answers on surveys
- Inaccurate measurements that we need to discard- we're mainly talking about easily detectable outliers here
- Canceled trials of an experiment

To do something about this, however, we need to figure out the following.

- What contributes to the *probability* of a data point being absent?
- Can this missing data be interpolated using the data we already have? Or not?

Just Deleting It

The easiest way to deal with this is just to delete all the tuples with any missing values, so we don't have any 'incomplete observations'. All we have to do in this case is just use `df.dropna()` to trop the appropriate row.

Be warned that a loss of a sample could lead to a variance greater than what's reflected by the size of our data. This could cause bias!

9 Footnotes

Taken by Akilesh Praveen.