Data 100 Notes

Evelyn Koo

Summer 2022

Contents

Introduction

Notes for Data 100, taken using Vim and LaTeX.

Remarks

Notes are taken from the slides of the Spring 2022 edition of Data 100.

Some of the figures may be messy or poorly drawn as this is my first time seriously learning TikZ.

2022-05-17

Lecture 1

Course Overview

1.1 Overview

1.1.1 What is Data Science?

"Data Science" part of the course name. Fundamentally an interdisciplinary field - it is the application of data-centric, computational, and inferential thinking to understand the world (science) and solve problems (engineering).

Good data analysis is not just simply applying a statistics recipe or using a software. While there are many tools for data science, they are just tools - they do not do the important thinking.

Example Questions in Data Science

- What show should we recommend to our user to watch?
- In which markets should we focus our advertising campaign?
- Is the use of the COMPAS algorithm for prison sentencing fair?

1.1.2 What will you learn in this class?

"Principles and Techniques" part of the course name. Course goals:

- **Prepare** students for advanced Berkeley courses in data management, ML, and statistics by providing the necessary foundation and context.
- **Enable** students to start careers as data scientists by providing experience working with real-world data, tools, and techniques.
- **Empower** students to apply computation and inferential thinking to address real-world problems.

Topics Covered

- **Data Analysis and Visualization**: Using NumPy, Pandas, visualization with matplotlib, Seaborn, plotly; Relational Databases and SQL, Regex
- **Statistics**: Sampling, probability, and random variables.

- **Machine Learning/Statistics Techniques**: Model designing/loss formulation, linear regression, feature engineering, regularization, bias-variance tradeoff, cross-validation; gradient descent, logistic regression; decision trees and random forests, PCA.

1.2 Data Science Lifecycle

High-level description of the data science workflow.

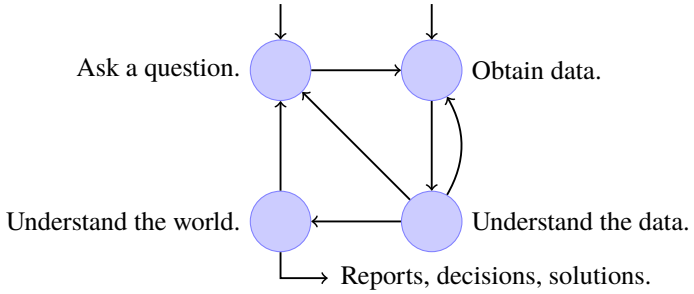


Figure 1: Diagram of the data science lifecycle. Note the two different entry points.

1. **Question/Problem Formation** (Ask a question):
   - What do we want to know?
   - What problems are we trying to solve?
   - What are the hypotheses we want to test?
   - What are our metrics for success?
2. **Data Acquisition and Cleaning** (Obtaining Data):
   - What data do we have and what data do we need?
   - How will we sample more data?
   - Is our data representative of the population we want to study?
3. **Exploratory Analysis and Visualization** (Understand the Data):
   - How is our data organized and what does it contain?
   - Do we already have relevant data?
   - What are the biases, anomalies, or other issues with the data?
   - How do we transform the data to enable effective analysis?
4. **Prediction and Inference** (Understand the World):
   - What does the data say about the world?
   - Does it answer our questions or accurately solve the problem?
   - How robust are our conclusions and can we trust the predictions?

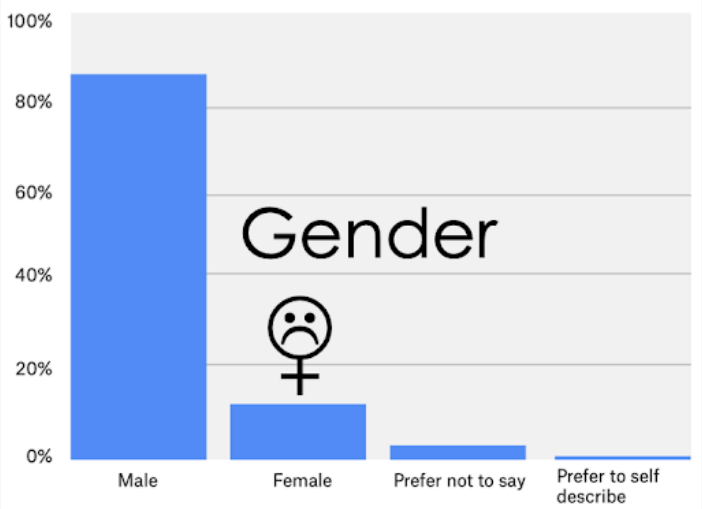
Example 1.1: Demo: Data Science Lifecycle

Begin with asking a question (step 1): Who are you?

Then gather and clean data (steps 2 and 3). Our population is Data 100 students, and some sub-questions could

be # of students, majors, year, diversity.

Focus on the final sub-question - diversity. Surveys of data scientists suggest that there are far fewer women:



We want to see if this holds in the class (steps 4 and 1). We now have a new question: "What fraction of students are female?", and the process repeats itself again.

Now, since we want to find female students, a new question arises: "Can we estimate a person's sex using their name?" and we obtain more data: SSN baby names (steps 1, 2, and 3).

Using this data, we try to estimate the fraction of female students in the class (step 4). A possible classifier:

1. SSN: Proportion of female babies per name.
2. Use step 1 to classify each student name as F, M, or unknown.
3. Average step 2 to get the class proportion of females.

However, the current model doesn't account for uncertainty. Create a new classifier:

1. SSN: Proportion of female babies per name.
2. For each student name from above:
   (a) Pick a number in $[0, 1)$.
   (b) If 2a is less than the SSN proportion (or 0.5 for Unknown), classify student as F; else M.
3. Average step 2 to get a class proportion of females.

Recap

- Find Data 100 data.
- Explore interesting things about the class: names, majors, etc.
  - Get stuck on a question: gender diversity.
- Find more data: US SSN baby names.
  - *Approximate* gender with sex.

- Create a classifier:
  - Simple classifier: classifies names as exactly F/M.
  - Random classifier: all names have some probability of F.

Limitations

- US name data - while there are students from around the world here at Berkeley.
- Names from since 1937; however, most students are born around 2000.
- No "rare" names.
- Sex as a proxy for gender - gender has been proxied to a binary classification.

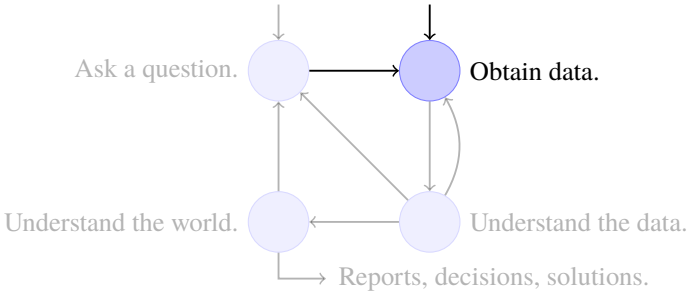
Class data was *fundamentally insufficent* to answer original question. Currently don't have the data to answer the question. Can survey the students to get the true proportion or use the data we have to get an estimate.

2022-05-17

Lecture 2

Data Sampling and Probability

Previously, we mentioned the data science lifecycle. Today, we focus on step 2: collecting data. How do we collect data?



2.1 Sampling

2.1.1 Censuses and Surveys

US decennial census - last held in April 2020. Counts *every person* living in all 50 states, DC, and US territories - not just citizens. Mandated by the constitution and participation is required by law.

Important uses of the census:

- Allocation of federal funds.
- Congressional representation (update based on population).
- Drawing congressional/state representative districts.

Definition 2.1: Census

In general, a census is an official count or survey of a population, typically recording various details of individuals.

Definition 2.2: Survey

A survey is a set of questions; e.g. workers survey individuals and households.

What and how the questions are asked can affect how the respondent answers or whether they answer at all.

2.1.2 Sampling: Definitions

While a census is great, it is expensive and difficult to execute (e.g. would all voters be willing to participate in a voting census before an election?). For this, we can use samples.

Definition 2.3: Sample

A sample is a subset of the population.

- Often used to make inferences about the population.
- How the sample is drawn will affect its accuracy.
- Common sources of error:
  - **Chance error**: random samples vary from what is expected in any direction.
  - **Bias**: systematic error in one direction.

Population, Sample, and Sampling Frame

- **Population**: group you want to learn something about.
- **Sampling Frame**: list from which the sample is drawn (e.g. if you are sampling people, the sampling frame is the set of all people who can end up in the sample).
- **Sample**: who you actually end up sampling; a subset of the sampling frame.

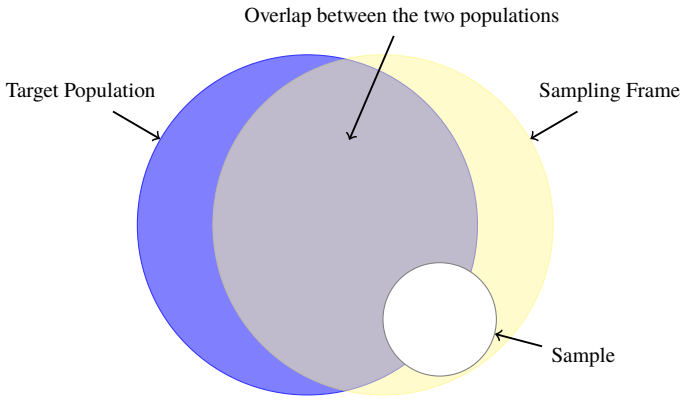


Figure 2: Target Population, Sampling Frame, and Sample. Note that there may be people in the sampling frame (and therefore, the sample) which are not part of the population.

2.1.3 Bias: A Case Study

1936 Presidential Election

Here, President FDR went up for re-election against Alf Landon. Polls were conducted in the months leading up to the election trying to predict the outcome.

One of the polls was from The Literary Digest, a magazine which had successfully predicted the outcome of 5 general elections up till 1936. Their survey was sent to 10 million people from phone books, magazines subscribers, and country club members. This poll got that 43% of people would choose FDR in the election, but in the actual election, 61% of people chose FDR.

Literary Digest poll: What went wrong?

1. Sampling frame was not representative of the US population.
    - Chose people who had phone numbers and subscribed to their magazines or country clubs.
    - Such people tend to be more affluent and more likely to vote Republican.
2. Non-response: only 2.4 million people (24%) filled out the survey. Who knows how the other 76% would have responded?

Gallup's Poll

In addition to the Literary Digest poll, George Gallup also made predictions about the 1936 elections. Looking at the table, his result (56%) was much closer to the actual election and with a smaller sample size.

	% Roosevelt	# surveyed
Election	61%	All voters (~45 million)
Literary Digest Poll	43%	10 million
Gallup's Poll	56%	50 000
Gallup's prediction of Digest's Poll	44%	3 000

Gallup was even able to predict the Literary Digest's prediction within 1% using a much smaller sample size. Why?

- He predicted their sampling frame (phonebook, magazine subscribers, country clubs).
- So he sampled the same individuals.

Common Biases

Samples, while convenient, are subject to chance error and bias.

- **Selection Bias**: Systematically avoiding/favoring particular groups. Avoid by examining the sampling frame and the method of sampling.
- **Response Bias**: People don't always respond truthfully. Avoid by examining the nature of questions and the method of surveying.
- **Non-response Bias**: People don't always respond. Avoid by making surveys short and being persistent - people who don't respond aren't like the people who do.

It's likely that the Literary Digest sample was a case of selection bias - they favored choosing those who they could easily access through the phonebook/subscriptions.

2.2 Probability

2.2.1 Probability Samples

When sampling, go for quality over quantity - don't just go for a big sample (as that can lead to having a big sample that is of poor quality).

Common Non-Random Samples

- **Convenience Sample**: choose whoever you can get ahold of (e.g. when trying to measure the weight of mice, choosing whichever mice you can get ahold of as samples to measure weight).
    - Not a good idea for inference.
    - Haphazard ≠ random.

  - Sources of bias can be present in ways that you may not think of (e.g. all the convenient samples can be of one group).

- **Quota Sample**: Specify desired breakdown of various subgroups in population, then choose to reach targets however you can (e.g. Sampling individuals in your town and having the sample match the age distribution from the census).
  - Reaching quotas "however you can" is not random.
  - Sample will represent some of the population (e.g. age), but not all of the population (e.g. gender, ethnicity, income may not be well represented in the sample).

So why sample at random? One thing is to reduce bias (despite this, random samples can still produce biased estimates); however, the main reason is that we can estimate bias and chance error with random samples, allowing us to quantify uncertainty.

Definition 2.4: Probability Sample

Sample from a random sampling scheme. Has the following properties:

- Must be able to provide the chance that any specified set of individuals will be in the sample.
- All individuals in the population need not have the same chance of being selected.
- Will be able to measure the errors since you know all the probabilities.

Example 2.1

Suppose there are 3 students: A, B, and D. Sample 2 students with the following procedure:

- Choose A with probability 1.
- Choose from $\{B, D\}$ with probability 0.5 each.

Possible Outcomes:

Subsets of 2	Probabilities
$\{A, B\}$	0.5
$\{A, D\}$	0.5
$\{B, D\}$	0

This is a probability sample, though not a great one:

- Of the 3 people in each population, know the probability (chance) of getting each subset.
- If we're measuring the average distance students live from campus, then problems arise:
  - The sampling frame does not account for the entire population - only three students.
  - Some *chance error* depending on if the sample is AB or AD.
  - Since we choose A w.p. 1, the sample *biases* towards A's response

Common Random Sample Schemes

- **Random Sample with replacement**: sample drawn uniformly at random with replacement.
- **Simple random sample (SRS)**: sample drawn uniformly at random without replacement.
  - Each individual/subset of individuals has the same chance of being selected.
  - Each pair has the same chance as every other pair; each triple has the same chance as every other triple (independent).

Note 2.1

Random doesn't always mean "uniformly at random", but with these examples, it does.

Example 2.2

Consider the following sampling scheme:

- Class roster has 1100 students listed alphabetically.
- Pick one of first 10 students on the list at random.
- Given the number from previously, sample the student and every 10th student listed after that (e.g. students 8, 18, 28, ...).

Determine the following:

1. Is this a probability sample?
2. Does each student have the same probability of being selected?
3. Is this a simple random sample?

1. Yes, we can provide the probability of each subset being elected, e.g. $\mathbb{P}(S = \{8, 17, \ldots\}) = 0$, $\mathbb{P}(S = \{8, 18, \ldots\}) = \frac{1}{10}$.
2. Yes, each student has a 1/10 chance of being selected.
3. No, the samples are not independent - the probabilities are different for each pair. As noted above, $\mathbb{P}(\{8, 17\} \in S) = 0 \neq \mathbb{P}(\{8, 18\} \in S) = \frac{1}{10}$.

Often, in data science, the population is large, but we can only afford to sample a small number of individuals. If the population is relatvely large compard to the sample, then sampling with and without replacement are pretty much the same.

Example 2.3: Sampling with/without Replacement

Suppose there are 10 000 people in the population; 7 500 who like Snack 1 and 2 500 who like Snack 2. Find the probability of all people in a sample of 20 liking Snack 1 with and without replacement.

- **SRS/Without Replacement**: first probability is $\frac{\#\text{ people who like S1}}{\text{Population}}$, then $\frac{\#\text{ people who like S1}-i+1}{\text{Population}-i+1}$ for the ith person.

$$P(\text{All 20 people like S1}) = \left(\frac{7500}{1000}\right)\left(\frac{7499}{9999}\right)\cdots\left(\frac{7482}{9982}\right)\left(\frac{7481}{9981}\right) \approx 0.03153.$$

- **With Replacement**: all probabilities are $\frac{\#\text{ people who like S1}}{\text{Population}} = \frac{3}{4}$.

$$P(\text{All 20 people like S1}) = \left(\frac{3}{4}\right)^{20} \approx 0.003171.$$

The results are close as the without replacement proportion remains close to the with replacement proportion; however, it is easier to calculate the probability without replacement.

Recap

If a sample was randomly sampled with replacement from a population:

- It is a probability sample.
- We can quantify error and bias.
- Given the population distribution, we can compute the probability of getting a particular sample.

Note 2.2

We almost never know the population distribution unless we take a census. However, framing allows us to quantify our uncertainty in any analysis/inference using our sample.

Example 2.4: Application: The Gallup Poll Today

tbc

2.2.2 Multinomial and Binomial Probabilities

Binomial and multinomial probabilities arise when we:

- Sample at random with replacement. (Think of as same trial)
- Sample a fixed number (n) times.
- Sample from a categorical distribution: 2 - binomial, > 2 - multinomial.

	Binomial	Multinomial
# Categories	2 categories	> 2 categories
Example	Bag of Marbles: 60% blue, 40% not blue	Bag of Marbles: 60% blue, 30% green, 10% red

Goal: count the number in each category that end up in our sample. `np.random.multinomial` returns these counts.

Example 2.5: Binomial Probability

Suppose we sample at random with replacement 7 times from a bag of marbles (60% blue - b, 40% not blue - n). Define the following events:

- A: Get the following marbles in the following order: bnbbbnn.
- B: Get 4 blue marbles and 3 not blue marbles.

Find $\mathbb{P}(A)$. Is $\mathbb{P}(B)$ greater than, the same as, or smaller than $\mathbb{P}(A)$?

$$\begin{aligned}\mathbb{P}(A) &= (0.6) \cdot (0.4) \cdot (0.6)^3 \cdot (0.4)^2 \\ &= (0.6)^4 (0.4)^3.\end{aligned}$$

$\mathbb{P}(B) > \mathbb{P}(A)$. This is since A is a specific case of B (4 blues, 3 not blues). B could have any permutation of the marbles as long as there are 4 blue marbles out of 7. Therefore, there are more outcomes under event B, and since each outcome is uniform, the probability of B (multiple outcomes) happening is greater than A (one outcome).

$$\mathbb{P}(B) = \left(\frac{7!}{4!3!}\right)(0.6)^4(0.4)^3 = \binom{7}{4}(0.6)^4(0.4)^3.$$

Example 2.6: Multinomial Probability

Sample with replacement 7 times from a bag of marbles (60% blue, 30% green, 10% red). Define the following events:

- A: Get the following marbles in the following order: bgbbbgr.
- B: Get 4 blue marbles, 2 green marbles, and 1 red marble.

To find A, use the product rule for independent marble draws:

$$\begin{aligned}\mathbb{P}(A) &= (0.6) \cdot (0.3) \cdot (0.6)^3 \cdot (0.3) \cdot (0.1) \\ &= (0.6)^4 (0.3)^2 (0.1)\end{aligned}$$

.

For B, the result is A (specific, ordered combination) by the number of outcomes with the exact marble distribution:

$$\mathbb{P}(B) = \left(\frac{7!}{4!2!1!}\right)(0.6)^4(0.3)^2(0.1).$$

Generalization of Multinomial Probability: For drawing with replacement n times from a population broken into m categories where $\sum_{i \in m} p_m = 1$ (i.e. each category has proportion p_i of the population). Then, the multinomial probability of drawing k_i individuals from each category i is:

$$\frac{n!}{k_1!k_2!\ldots k_m!} p_1^{k_1} p_2^{k_2} \cdots p_m^{k^m}.$$

Note 2.3: Derivation of Multinomial Probability Coefficient

Looking back at the previous example, how did we get $\frac{7!}{4!2!1!}$? Here are a few ways to think of it:

1. Start with 7 open positions, then choose 4, which is then $\binom{7}{4}$. Now 3 positions remain for green marbles, of which you choose 2, which is then $\binom{3}{2}$. Now, one position remains for red marbles, which you choose 1, which is then $\binom{1}{1}$.
$$\binom{7}{4} \cdot \binom{3}{2} \cdot \binom{1}{1} = \frac{7!}{4!3!} \cdot \frac{3!}{2!1!} \cdot \frac{1!}{1!0!} = \frac{7!}{4!3!1!}.$$
2. Start with 7 open positions, want to arrange marbles in as many ways as possible, which is 7! to begin with. However, since there are 4 blue marbles (identical, meaning putting blues in any 4 places would be just one arrangement), you can divide by 4!, and you can do the same for 2 greens (divide by 2!).

2022-05-20

Lecture 3

Pandas I

Tabular data is one of the most common data formats and the primary focus of Data 100. In Data 8, we used the `Table` class of the `datascience` library. Here in Data 100, we use the `DataFrame` class of the `Pandas` library.

[2]:
```
elections = pd.read_csv("elections.csv")
elections # if we end a cell with an expression or variable name, the result will print
```

[2]:

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	1832	Andrew Jackson	Democratic	702735	win	54.574789
...	...	...	...	...	...	...
177	2016	Jill Stein	Green	1457226	loss	1.073699
178	2020	Joseph Biden	Democratic	81268924	win	51.311515
179	2020	Donald Trump	Republican	74216154	loss	46.858542
180	2020	Jo Jorgensen	Libertarian	1865724	loss	1.177979
181	2020	Howard Hawkins	Green	405035	loss	0.255731

182 rows × 6 columns

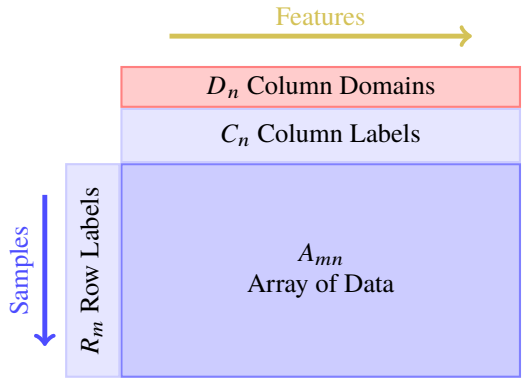


Figure 3: Left: A sample Pandas DataFrame. Right: A generic DataFrame with a statistical interpretation; rows are samples from the population, columns are features that each sample has.

Note 3.1: Pandas DataFrame API

Pandas DataFrame API (application programming interface, or set of applications supported by the class) is very large. When dealing with problems, Google them - they are often found in the documentation or stack overflow.

API Reference

3.1 Indexing

One of the most basic tasks of manipulating a DataFrame is to extract rows/columns. Since the Pandas API is very large, this means there are many ways to do things.

Additionally, there is a lot of "syntactic sugar" - methods that may be useful and lead to concise code that are not necessary for the library to function (e.g. `.head` / `.tail`, which select the first/last n rows).

3.1.1 head/tail

General Form: `df.(head|tail)(<num_rows>)`

Selects the first/last n rows. `df.head(5)` is equivalent to `df.loc[[0:4]]`, while `df.tail(5)` is equivalent to `df.loc[[-5:]]`.

3.1.2 loc

General Form: `df.loc[<rows>, <cols>]`.

Fundamentally, `loc` selects items by label - the bolded text in the top (column names/features) and left (row numbers/samples). Can pass in a list, slice, or single value into `loc`.

To select all columns, omit the second argument. To select all rows, pass in `:` as the first argument - it is equivalent to selecting the entire array of row numbers.

3.1.3 iloc

General Form: `df.iloc[<row-nums>, <col-nums>]`.

Fundamentally, `iloc` selects items by number. For rows, this is the same as the label; however, this is not always the case for columns. Like `loc`, `iloc` can have single values, slices (`[<start-num, end-num)`), or arrays passed in as arguments.

```
[9]: elections.loc[[87, 25, 179], ["Year", "Candidate", "Result"]]
```

[9]:

	Year	Candidate	Result
87	1932	Herbert Hoover	loss
25	1860	John C. Breckinridge	loss
179	2020	Donald Trump	loss

```
[10]: elections.loc[[87, 25, 179], "Popular vote":"%"]
```

[10]:

	Popular vote	Result	%
87	15761254	loss	39.830594
25	848019	loss	18.138998
179	74216154	loss	46.858542

```
[12]: elections.loc[[87, 25, 179], "Popular vote"]
```

```
[12]: 87     15761254
      25       848019
      179    74216154
      Name: Popular vote, dtype: int64
```

```
[10]: elections.loc[:, ["Year", "Candidate", "Result"]]
```

[10]:

	Year	Candidate	Result
0	1824	Andrew Jackson	loss
1	1824	John Quincy Adams	win
2	1828	Andrew Jackson	win
3	1828	John Quincy Adams	loss
4	1832	Andrew Jackson	win
...	...	...	...
177	2016	Jill Stein	loss
178	2020	Joseph Biden	win
179	2020	Donald Trump	loss
180	2020	Jo Jorgensen	loss
181	2020	Howard Hawkins	loss

182 rows × 3 columns

Figure 4: Left: selecting different values with `loc`. Right: selecting all rows with `loc`.

And just like `loc`, to select all rows, pass in `:`, and to select all columns, omit the second argument.

Note 3.2: loc vs. iloc

When deciding between `loc` and `iloc`, usually `loc` is the better choice - it's safer (e.g. will select the right columns if columns are mixed up) and more legible (e.g. `["colname1", "colname2"]` over `[1, 2]`).

However, `iloc` can still be useful - e.g. it would make more sense to index into the middle of a dataframe with `iloc`.

3.1.4 []

General Form: `df[(<row-nums>|col-labels|col-label)]`.

Unlike `loc` and `iloc`, `[]` only takes in one argument rather than rows or columns. The use of `[]` changes on the context - i.e., it is context sensitive.

- For a slice of numbers, it returns the numbered rows.
- For a list of column names or a single column name, it returns the column(s) in question.

Example 3.1

Suppose we have the following dataframe `weird`:

	0	1
0	topdog	topcat
1	botdog	botcat

What does the following return:

- `weird[1]`

```
[13]: elections.iloc[[1, 2, 3], [0, 1, 2]]
```

[13]:

	Year	Candidate	Party
1	1824	John Quincy Adams	Democratic-Republican
2	1828	Andrew Jackson	Democratic
3	1828	John Quincy Adams	National Republican

```
[14]: elections.iloc[[1, 2, 3], 0:2]
```

[14]:

	Year	Candidate
1	1824	John Quincy Adams
2	1828	Andrew Jackson
3	1828	John Quincy Adams

```
[15]: elections.iloc[[1, 2, 3], 1]
```

```
[15]: 1    John Quincy Adams
      2       Andrew Jackson
      3    John Quincy Adams
      Name: Candidate, dtype: object
```

```
[16]: elections.iloc[:, [0, 1, 4]]
```

[16]:

	Year	Candidate	Result
0	1824	Andrew Jackson	loss
1	1824	John Quincy Adams	win
2	1828	Andrew Jackson	win
3	1828	John Quincy Adams	loss
4	1832	Andrew Jackson	win
...	...	...	...
177	2016	Jill Stein	loss
178	2020	Joseph Biden	win
179	2020	Donald Trump	loss
180	2020	Jo Jorgensen	loss
181	2020	Howard Hawkins	loss

Figure 5: Left: selecting different values with `iloc`. Note how instead of column names, we used column numbers. Right: selecting all rows with `iloc`.

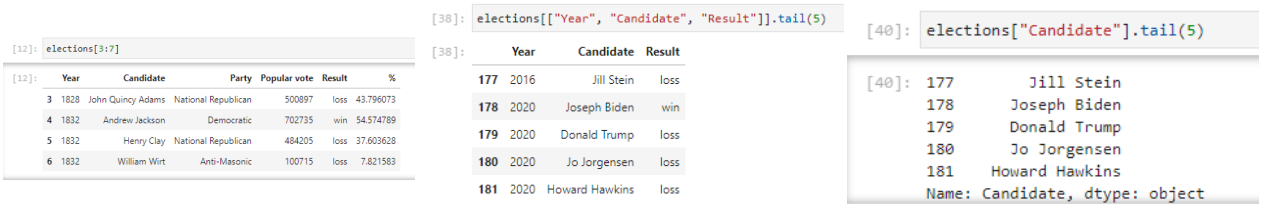


Figure 6: `[]` in different contexts.

- `weird["1"]`
- `weird[1:]`

- `weird[1]`: single column number, so it should return column 1 - i.e. the first column "0".

	0
0	topdog
1	botdog

- `weird["1"]`: single column label, so it should return column "1".

	1
0	topcat
1	botcat

- `weird[1:]` : row slices, so just the first row and beyond.

	0	1
1	botdog	botcat

Note 3.3: `[]` Usage and Dot Notation

Whereas `loc` and `iloc` both accept multiple arguments, `[]` does not and selects based on context. Therefore, the syntax is more precise and preferred for real world practice compared to `loc` .

Another way to index is with dot notation; more info can be found here.

3.2 DataFrames, Series, and Indices

Notice in the previous examples above, if we select only a single column, the notebook returns a different format. This is since the return output is a Series and not a DataFrame.

```
type(elections) # DataFrame
type(elections["Candidate"]) # Series
```

This leads into the Pandas data structures. Here are the three fundamental data structures in Pandas:

- **DataFrame**: 2D Tabular Data.
- **Series**: 1D single-column/columnar data.
- **Index**: sequence of row labels.

We can think of a dataframe as a collection of series that all have the same index.

Indices are not necessarily row numbers; they can also be non-numeric and have names, e.g. State. They also do not have to be unique.

However, column names are almost always unique. (e.g. it would not make sense to have two columns with the same name, but you can force duplicate columns to have them have the same column name).

If you want row/column labels: you can use `df.index` for row labels and `df.columns` for column labels.

Note 3.4: Getting a DataFrame instead of a Series

If you want a DataFrame instead of a Series input, there are two ways to do so:

- `Series.toFrame()` method.
- Enclose the single column of interest in a list (e.g. `df[[<col-name>]]` as opposed to `df[col-name]`)

3.3 Conditional Selection

`loc` also supports Boolean array input (usually generated by using logical operators on Series). This can be combined with operators, allowing for filtering with multiple criteria

```
elections[elections["Party"] == "Independent"] # checks the Party series for each entry, keeps the entries which the party is independent

elections[(elections["Result"] == "win") & (elections["%"] < 47)] # checks if the election result is a win and if the winning percent is less than 47%
```

Example 3.2

Which of the following statements returns a DataFrame of the first 3 candidate names only for candidates that won more than 50% of the vote?

Using `loc`: select all the candidates (rows) that have an election percent > 50, choose candidate, year columns, first three rows that satisfy the condition (head).

End Code: `elections.loc[[elections['%'] > 50, ["Candidate", "Year"]].head(3)`

Using `iloc`: The columns in question are candidate (0) and Year (3), and the first three rows that satisfy the conditions are 0, 3, 5. Therefore, the end code should be `elections.iloc[[0, 3, 5], [0, 3]]`.

While boolaen array selection is useful, it can lead to overly verbose code when there are complex selections. There are many alternative functions to make the code more concise:

- `.isin`: returns True if the row value is in an array of results.
- `.str.startswith`: return True if the row value starts with the string provided.
- `.query`: Similar to SQL queries select keyword (select rows which fit the criteria). can access Python variables with `@<var-name>`.
- `.groupby.filter`: to be discussed in the next lecture.

```
# original, verbose code
elections[(elections["Party"] == "Anti-Masonic")  |
          (elections["Party"] == "American")      |
          (elections["Party"] == "Anti-Monopoly") |
          (elections["Party"] == "American Independent")]

# .isin
a_parties = ["Anti-Masonic", "American", "Anti-Monopoly", "American Independent"]
elections[elections["Party"].isin(a_parties)]
# .str.startswith
elections[elections["Party"].str.startswith("A")]
# query - select rows of elections after 2000 and the result is a win
elections.query('Year >= 2000 and Result == "win"')
# query with @
parties = ["Republican", "Democratic"]
elections.query('Result == "win" and Party not in @parties') # select winners that don't belong to republican/democratic parties
```

3.4 Utility Functions

Pandas Series/DataFrames support mathematical, NumPy, and built-in functions as long as the data is numerical. Additionally, Pandas has its own utility functions:

- `size/shape`: size returns the amount of data entries (rows by columns), while shape returns the shape (rows, columns).
- `describe`: gives a brief description of the numerical data, returning the count, mean/stdev, and 5 points (min, 25%, 50%, 75%, and max).

- `sample` : samples a random selection of rows without replacement (add `replace=True`) for replacement. Can be chained with other methods and operators (e.g. `query` , `iloc` , etc.).
- `Series.value_counts` : counts the number of occurences of each unique value (returned as a series of value-count).
- `Series.unique` : returns an array of all unique values in a Series.
- `sort_values` : sorts values in a Series or a DataFrame (must list column on which to sort for a DataFrame) in ascending order (for descending order, set `ascending=False`).

2022-05-21

Lecture 4

Pandas II: Grouping, Aggregation, Pivot Tables Merging

4.1 Adding, Modifying, and Removing Columns

Previously, we saw how we could find the most popular male names in California using `query` and sorting the result.

```
babynames.query("Sex ==M and Year == 2020").sort_values("Count", ascending=False) # chooses only rows
↪  from 2020 which are male, then sort in descending order
```

	State	Sex	Year	Name	Count
391409	CA	M	2020	Noah	2608
391410	CA	M	2020	Liam	2406
391411	CA	M	2020	Mateo	2057
391412	CA	M	2020	Sebastian	1981
391413	CA	M	2020	Julian	1679
...	...	...	...	...	...
393938	CA	M	2020	Gavino	5
393937	CA	M	2020	Gaspar	5
393936	CA	M	2020	Gannon	5
393935	CA	M	2020	Galen	5
394178	CA	M	2020	Zymir	5

However, how can we find the longest names? If we sort by the names column, it will just return the names in reverse alphabetical order. After some searching, it turns out you can add in a `key` parameter which tells Pandas how to sort the values in the column.

```
# start process - sorts names by alphabetical order
babynames.query("Sex == M and Year == 2020").sort_values("Name", ascending=False)
# new code - add key of baby name length
babynames.query("Sex == M and Year == 2020").sort_values("Name", key=lambda name: name.str.len(),
↪  ascending=False)
```

However, this wasn't a feature until 2020. Prior to that, we would need to come up with another solution:

- Create a temporary column getting the length of the baby name.

	State	Sex	Year	Name	Count
393226	CA	M	2020	Zyon	9
394178	CA	M	2020	Zymir	5
393352	CA	M	2020	Zyan	8
392118	CA	M	2020	Zyaire	37
392838	CA	M	2020	Zyair	13
...	...	...	...	...	...
393106	CA	M	2020	Aamir	9
393822	CA	M	2020	Aalam	5
393354	CA	M	2020	Aaditya	7
393353	CA	M	2020	Aadi	7
392994	CA	M	2020	Aaden	10

	State	Sex	Year	Name	Count
393478	CA	M	2020	Michaelangelo	7
393079	CA	M	2020	Michelangelo	10
392964	CA	M	2020	Maximilliano	11
394047	CA	M	2020	Maxemiliano	5
392610	CA	M	2020	Maximillian	16
...	...	...	...	...	...
393110	CA	M	2020	Aj	9
392856	CA	M	2020	Cy	12
393558	CA	M	2020	An	6
393981	CA	M	2020	Jj	5
392750	CA	M	2020	Om	14

Figure 7: The first approach only yields the names in reverse alphabetical order, but sorting with the `key` parameter yields the desired result.

  - Create a new series with only lengths.
  - Add the series to the dataframe.
- Sort using that column.
- Drop the temporary column. (General Form: `df.drop(<name>, (axis="columns"))`)

```
# creating a new column
babynames["name_lengths"] = babynames["Name"].str.len() # new series - applies len to the Name series, then assigns the series "name_lengths" to the dataframe as a column
babynames = babynames.sort("name_lengths", ascending=False) # sort by name_length columns in descending order
babynames = babynames.drop("name_lengths", axis="columns") # drop the column - add axis = "columns" to specify it is a column (default drop rows)
```

4.1.1 Sorting by Arbitrary Functions

To sort by arbitrary functions (e.g. # occurrences of "dr" and "ea" in a string), use `Series.map` method. (Think of the `tbl.apply()` method in datascience)

```
def dr_ea_count(string):
    return string.count("dr") + string.count("ea")

babynames["dr_ea_count"] = babynames["Name"].map(dr_ea_count) # Series.map(<func>)
babynames.sort_values("dr_ea_count", ascending=False)
```

4.2 Group By

Let's try to find the female baby name whose popularity has fallen the most using RTP (ratio to peak - current number over peak number in a year).

e.g. for the name "Jennifer" measured in 2020, there were 172 Jennifers over the peak of 6064 Jennifers in 1972. Therefore, the RTP is $\frac{172}{6064} = 0.0233$.

	State	Sex	Year	Name	Count	dr_ea_count
108712	CA	F	1988	Deandrea	5	3
293396	CA	M	1985	Deandrea	6	3
101958	CA	F	1986	Deandrea	6	3
115935	CA	F	1990	Deandrea	5	3
131003	CA	F	1994	Leandrea	5	3

Figure 8: Graph of the amount of Jennifers each year.

Here's how to do it in Pandas:

```
max_j = max(babynames.query("Name == 'Jennifer' and Sex == 'F'")["Count"]) # max on the count series based on the query (females born named Jennifer)
cur_j = babynames.query("Name == 'Jennifer' and Sex == 'F'")["Count"].iloc[-1] # same query, select count series, choose the last row
rtp = cur_j / max_j # find the rtp - current vs max

# simplifying
def ratio_to_peak(series):
    return series.iloc[-1] / max(series) # current / max

j_counts = babynames.query("Name == Jennifer and Sex == 'F'")["Count"]
ratio_to_peak(j_counts) # returns the same results as the rtp variable
```

What about finding the RTP for every name and not Jennifer? A possible way is to create a dictionary of RTPs and get the RTP for each unique name.

```
#build dictionary where each entry is the rtp for a given name, e.g. rtps["jennifer"] should be 0.0231
rtps = {}
for name in babynames["Name"].unique(): # use Series.unique to get an array of unique values in a series
    counts_of_current_name = female_babynames[female_babynames["Name"] == name]["Count"] # choose the rows corresponding to the correct name
    rtps[name] = ratio_to_peak(counts_of_current_name)

#convert to series
rtps = pd.Series(rtps)
```

However, this approach is very slow and more complicated. The next method takes advantage of `.groupby` and `.agg`.

```
female_babynames.groupby("Name").agg(ratio_to_peak) # group by column Name, aggregate through the rtp function
# data 8 approach
female_babynames.group("Name", ratio_to_peak)
```

Note 4.1

The second method (using `.groupby` and the Pandas API) is preferred and the one you should be using - you should never be writing code with loops or list comprehensions on Series.

Note 4.2

Make sure to drop invalid columns before calling `.agg` - not doing so will result in errors.

Previously, it would display 1.0 for all invalid columns.

```
female_babynames.groupby("Name").agg(ratio_to_peak) # without dropping cols
female_babynames.groupby("Name")[["Count"]].agg(ratio_to_peak) # select Count column
```

Without Dropping Columns:

Name	Year	Count
Aadhira	1.0	0.600000
Aadhya	1.0	0.720000
Aadya	1.0	0.862069
Aahana	1.0	0.384615
Aahna	1.0	1.000000
...	...	...

Dropping Columns:

Name	Count
Aadhira	0.600000
Aadhya	0.720000
Aadya	0.862069
Aahana	0.384615
Aahna	1.000000
...	...

Renaming Columns

Use `.rename` (General Form: `df.rename(columns= \{<old colname1>: <new colname1>, ..., <old-colnameN>: <new-colnameN> \})`) to rename columns.

```
rtp_table = female_babynames.groupby("Name")[["Count"]].agg(ratio_to_peak)
rtp_table = rtp_table.rename(columns = {"Count": "Count RTP"}) # change colname using a dict
```

Example 4.1

Write a `groupby.agg` call that returns the total number of babies with the same name.

As each entry records the name, year, and the count of babies born that year with the name, we should group by name and take the count column, then aggregate by the sum of the count in each year.

```
ex1 = female_babynames.groupby("Name")[["Count"]].agg(sum)
```

Example 4.2

Write a `grouby.agg` call that returns the total number of babies born per year.

Again, each entry records name, year, and count of babies born that year. Therefore, it makes sense to group by year and aggregate by the total count for each entry in each year.

```
ex2 = female_babynames.groupby("Year")[["Count"]].agg(sum)
```

Note 4.3: Shorthand groupby methods

Pandas has shorthand functions to use in place of `agg`, e.g. instead of `.agg(sum)`, you can use `.sum()`.

For more reference, check the Pandas reference here.

4.2.1 Quick Look at EDA

If we plot the table, we get the following graph of babies born per year. Does this say anything about the birth rate?

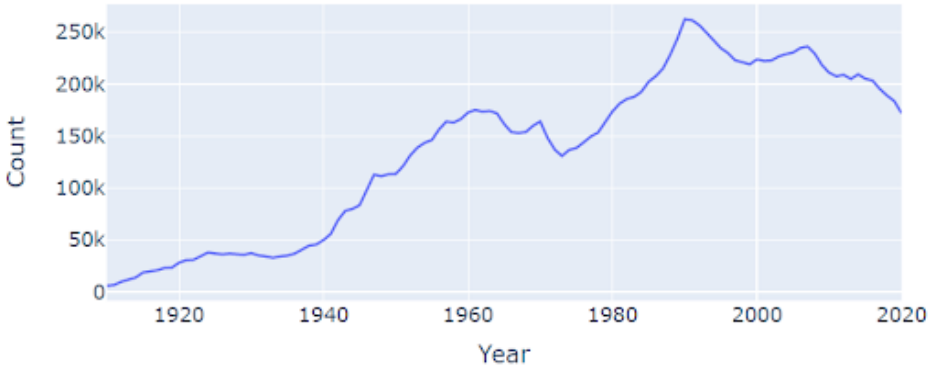


Figure 9: Graph of the dataframe as the birthrate.

However, the birthrate in 2020 is closer to 400k instead of 200k. What went wrong? Upon checking the data, we find some biases and issues. We only used female babies, not all babies are registered for social security, and the database does not have some of the more uncommon names (e.g. ones that appear fewer than 5 times per year). Therefore, it's important to look at the data and understand it as opposed to using it blindly.

Example 4.3

Why does the following table claim that Woodrow Wilson won the presidency in 2020? The function call is as follows:

```
elections.groupby("Party").agg(max).head(10)
```

This groups by party and aggregates the max value. However, it aggregates the max value for each column. The

max year would be the most recent - 2020, and Woodrow Wilson's name is last alphabetically, so it is last in the party, leading to this false result.

Follow up: Write code that returns the best percent result in each party.

Start by sorting the percent results. Then, we want to group by the party and return the first value in the percent results table.

```
top_results = elections.sort_values("%", ascending=False)
top_results_party = top_results.groupby("Party").agg(lambda x: x.iloc[0]) # gets the first row of each party group
```

Note 4.4

In Pandas, there are multiple ways to get the same result - each way has tradeoffs in terms of readability, performance, memory, and complexity. it may take a while to understand these tradeoffs, so a general method is to change appraoches if one becomes too convoluted.

4.2.2 Other groupby Features

So far, we've only used `.agg`, which organizes all rows of the same group into a subframe for that group, then creates a new dataframe with each group as a row and combines the values with the given function.

```
df.groupby("year").agg(sum) # each year split into a subframe for that year, new df with each year as a row, the value output is the sum of all the subframe values
```

However, if you just call `groupby`, then you end up with a `DataFrameGroupBy` object, which can be combined with many functions (other than `.agg`) to generate DataFrames. Some choices include the following:

- `.agg`: creates a new dataframe with one aggregated row per subframe.
- `.max`: creates a new dataframe aggregated with the max function.
- `.size`: creates a new series with the size of each subframe.
- `.filter`: creates a copy of the original dataframe, but only keeps the rows that satisfy the filter function.

More methods can be found on the Pandas reference here. (add some tikz/figures/tables)

4.3 Pivot Tables

If we want to group by multiple columns (e.g. using the babynames table, group by year and sex), we can group by both columns of interest. This creates a multi-indexed dataframe.

Another approach (from Data 8) is to pivot instead using the `pivot_table` function.

(General Use: `df.pivot_table(index=<col1>, columns=<col2>, values=[<val-cols>], aggfunc=<group-op>)`)

```
babynames.groupby(["Year", "Sex"]).agg(sum).head(6) # group with multiple cols
babynames.pivot_table(index="Year", columns="Sex", values=["Count"], aggfunc=np.sum) # pivot approach: sum up the count for each year/sex combination
```

		Count
Year	Sex	
1910	F	5950
	M	3213
1911	F	6602
	M	3381
1912	F	9803
	M	8142

	Count	
Sex	F	M
Year		
1910	5950	3213
1911	6602	3381
1912	9804	8142
1913	11860	10234
1914	13815	13111
1915	18643	17192

You can also have pivot tables with multiple columns.

```
babynames.pivot_table(index=["Year", "Name"], columns="Sex", values=["Count"], aggfunc=np.max)
```

	Count		Name	
Sex	F	M	F	M
Year				
1910	295	237	Yvonne	William
1911	390	214	Zelma	Willis
1912	534	501	Yvonne	Woodrow
1913	584	614	Zelma	Yoshio
1914	773	769	Zelma	Yoshio
1915	998	1033	Zita	Yukio

Figure 10: Pivot with multiple columns. Adds another pivoted column like the column present in the previous pivot table.

4.4 Joining Tables: An Overview

Suppose we want to know the popularity of presidential names among male names. To do this, we need to join tables using `pd.merge` .

(General Usage: `df.merge(left=<table1>, right=<table2>, left_on=<col1>, right_on=<col2>)`)

```
# creating tables - table containing baby names and candidate names
male_2020_babynames = babynames.query('Sex == "M" and Year == 2020')
elections["First Name"] = elections["Candidate"].str.split().str[0]
merged = pd.merge(left = elections, right = male_2020_babynames, left_on = "First Name", right_on =
↪  "Name") # merge the two tables, left_on/right_on is the common column
```

2022-06-15

Lecture 5

Data Wrangling, Exploratory Data Analysis

Previously, we worked with Pandas. Now, we have our data - what do we do with it next? Work to understand the data. This is split into processing and visualizing/reporting data.

5.1 Data Wrangling and EDA: An Infinite Loop

[Infinite Loop of Data Science - insert graphic]

Definition 5.1: Data Wrangling

Process of transforming raw data to facilitate subsequent analysis.

Often addresses issues such as the following:

- Structure/formatting
- Missing/corrupted values
- Unit conversion
- Encoding text as numbers

Data cleaning is a big part of data science.

Definition 5.2: Exploratory Data Analysis

"Getting to know the data", or the process of transforming, visualizing, and summarizing data to achieve the following:

- Build/confirm understanding of the data and its provenance (origin of data/methodology by which the data was produced).
- Identify/address potential issues in data.
- Inform the subsequent analysis.
- Discover potential hypotheses (be careful).

EDA is open-ended analysis; be willing to find something surprising.

John Tukey, mathematician and statistician from Princeton, who came up with Exploratory Data Analysis (as well as the idea of a bit and FFT), states that EDA is like detective work: "Exploratory data analysis is an attitude, a state of flexibility, a willingness to look for those things that we believe are not there, as well as those that we believe to be there."

5.2 Key Properties to Consider in EDA

What should we look for in EDA?

- **Structure:** "shape" of a data file.
- **Granularity:** how fine/coarse each datum is.
- **Scope:** how (in)complete the data is.
- **Temporality:** How the data is situated in time.
- **Faithfulness:** How well the data captures "reality".

5.2.1 Structure

Structure deals with the "shape" of a data file. Some major factors are: file format, file type, and multiple files (primary/foreign keys).

File Format

We prefer rectangular data for data analysis:

- Regular structure makes it easy to manipulate and analyze.
- A big part of data cleaning is to make data more rectangular.

(add records/rows and fields/columns tikz graphic)

Two types of rectangular data are tables and matrices:

- **Tables:** (e.g. dataframes in Python, SQL relations) named columns with different types, manipulated with data transformation languages (map, filter, group by, join).
- **Matrices:** Numeric data of the same type (float, int, etc.), manipulated with linear algebra.

These are usually formatted with TSV (tab separated values), CSV (comma separated values), or JSON.

Example 5.1

Suppose we are given a CSV of SF restaurant food safety scores. How do we understand the data structure?

Look for the size and format of the data file, and then look to find out how to read the data file into pandas. In the case of a CSV:

- Rows are delimited by a new line (`\n`).
- Columns are delimited by commas (`,`).

We can then use `pd.read_csv` .

Example 5.2: Reading a TSV vs. CSV

Now suppose the data was in a TSV? How can we read it, and what are the downsides of CSV/TSV data?

This time, instead of columns being delimited by commas, they are delimited by tabs (`\t`). Therefore, add the argument `delimiter="\t"` into the `pd.read_csv` function to read a TSV.

However, since these are marked by certain delimiters, having said delimiters in the data can cause issues (e.g. commas in data entries in a CSV, tabs in a TSV).

Example 5.3

Suppose we are given another dataset, which is a JSON of Berkeley covid cases by day. How can we understand the data's structure?

JSON is a less common format:

- Similar to Python dictionaries.

- Has strict formatting around quoting, which resolves issues in CSV/TSV.
- Saves metadata (data about data) along with records in the same file.
- However, it is not rectangular, and each record can have different fields.
- Additionally, nesting means records can contain tables, making things more complicated.

To sort out tabular data, find records using Python, and then use `pd.DataFrame` function.

Example 5.4: Mauna Loa Example

The Mauna Loa data is presented in CSVs. Which function do we use to load the data into Pandas?

Since it is a CSV, we can use `pd.read_csv` .

Note 5.1

Files may have mixed file formats, incorrect/missing extensions. This means you may need to explore the raw data file.

While in Data 100, we work primarily with CSV files, there are also other types of non-tabular data files elsewhere.

- **XML (Extensible Markup Language):** contains a nested structure, sort of like HTML.
- **Log Data:** Usually in txt and logs actions. Usually does not have a certain format, so you may need to make your own custom parser.

Variable Type

All data, regardless of format, is composed of records. Each record has a set of variables/fields. (For tabular data, records are rows, fields are columns; for non-tabular data, create records and wrangle into fields)

Variables are defined by the following:

- **Storage Type:** how they are stored in pandas (e.g. int, float, boolean, object, etc). Can be found with `df[colname].dtype` .
- **Feature Type:** conceptual notion of the information; uses expert knowledge and requires exploration of the data and possibly consulting the data codebook (if it exists).

(Insert variable feature types figure)

Var -> quantitative / qualitative -> discrete/continuous -> ordinal/nomial

Example 5.5

What is the feature type for each variable?

	Variable	Feature Type
1	CO2 Level (PPM)	Continuous
2	No. of Siblings	Discrete
3	GPA	Continuous
4	Income Bracket (low/med/high)	Ordinal
5	Race	Nominal
6	# Years of Education	Discrete
7	Yelp Rating	Ordinal

For Yelp rating, while it seems like it could be quantitative, it should be qualitative as it is usually presented with histograms (like qualitative data).

Multiple Files (Primary and Foreign Keys)

Sometimes, data comes in multiple files, and data will reference other data.

Definition 5.3: Primary Key

Column/set of columns in a table that determine the values of the remaining values.

Primary keys are unique. Some examples of primary keys are SSNs, IDs.

Definition 5.4: Foreign Keys

Columns/sets of columns that reference primary keys in other tables.

May need to join tables with `pd.merge`.

5.2.2 Granularity, Scope, Temporality

Granularity

- What does each record represent? (e.g. purchase, person, group of users)
- Do all records capture granularity at the same level? (e.g. some data will include summaries/rollups as records)
- If the data are coarse, how are the records aggregated? (e.g. sampling, averaging)

Scope

- Does my data cover my area of interest? (e.g. interested in studying crime in CA, but only have Berkeley data)
- Are my data too expansive? (e.g. interested in DS100 student grades, but have grades for entire stats dept classes; if the data is a sample, the filtered data could have poor coverage)
- Does my data cover the right time frame? (see Temporality)
- Recall: sampling frame is the population from which data were sampled; however, it may not be the population of interest. Want to know: how (in)complete is the frame and its data? (How is it situated in place/how well does the frame or data capture reality/how is the frame or data situated in time?)

Temporality

- **Data Changes:** When was the data collected/last updated?
- **Periodicity:** Are there patterns (e.g. diurnal/24h patterns)?
- Meaning of date/time fields? (e.g. when the event happened, when the data was collected/entered into the system, when the data was copied into a database)

Time also depends on time zones/daylight sayings (use Python `datetime` and Pandas `dt`). Another point to consider is different string representations of dates (varies by region).

There may also be some null values. Time is often measured in seconds since Jan 1, 1970, following UTC time.

5.2.3 Faithfulness and Missing Values

Faithfulness: Do I trust this data?

- Does my data contain unrealistic or "incorrect" values? (e.g. dates in the future for past events, nonexistent locations, negative counts, misspellings, large outliers)
- Does my data violate obvious dependencies? (e.g. age and birthday don't match)
- Was the data entered by hand? (e.g. spelling errors, fields shifted, did the form require all fields or provide default values)
- Obvious signs of data falsification? (e.g. repeated names, fake looking emails, repeated use of uncommon names/fields)

Common Problems and Solutions

- **Truncated Data:** e.g. early Microsoft Excel limits (65536 rows, 255 cols). Solution: be aware of consequences in analysis (e.g. how did truncation affect the sample?)
- **Timezone Inconsistencies:** convert to common timezone (e.g. UTC, timezone of the location for modeling behavior).
- **Duplicated Records/Samples:** identify/eliminate using primary key, understand implications on sample.
- **Spelling Errors:** Apply corrections/drop records not in dictionary, understand implications on sample.
- **Inconsistent/Unspecified Units:** Infer units, check that the values are in reasonable ranges for data.

Missing Data

Addressing missing data/default values:

- **Drop:** drop records with missing values. Most common approach. Check for biases induced by dropped values; missing/corrupt values may be related to something of interest.
- **Inputation:** Inferring missing values.
  - **Average Imputation:** replace with average value (usually closest related subgroup mean).
  - **Hot Deck Imputation:** replace with random value (random value from closest related subgroup mean).
- Drop missing values, but check for induced bias using domain knowledge.
- Directly model missing values using future analysis.

5.3 EDA Demo: Mauna Loa CO2

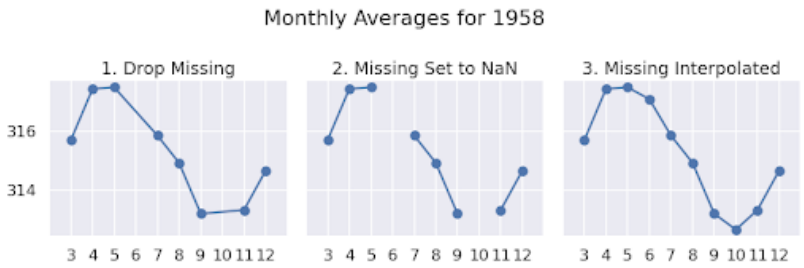
2022-06-25

Lecture 6

Regular Expressions

Previously, in Mauna Loa example, we worked with numeric data that had missing values. There were 3 options:

- Drop missing records.
- Use NaN (not a number) for missing values.
- Impute missing values with the interpolated `Int` column.



With numerical data, you do data wrangling with EDA, but with text data, wrangling is upfront and requires Python string manipulation and Regex.

Why work with text?

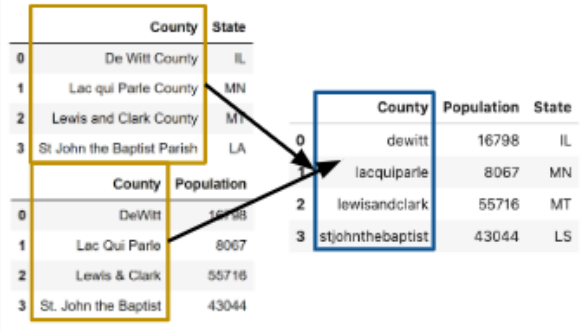
1. **Canonicalization:** Convert data that has more than one possible presentation into a standard form (e.g. join tables with mismatched labels)
2. **Extract** information into a new feature (e.g. extract dates and times from log files).

6.1 Python String Methods

A possible way we can work with text data is with Python string methods to parse/split/replace strings. Here are a few examples.

Example 6.1: Canonicalization

How do we make all the county names standardized (lower case, no spaces, remove punctuation, etc.)?



	County	State
0	De Witt County	IL
1	Lac qui Parle County	MN
2	Lewis and Clark County	MT
3	St John the Baptist Parish	LA

	County	Population
0	DeWitt	16798
1	Lac Qui Parle	8067
2	Lewis & Clark	55716
3	St. John the Baptist	43044

	County	Population	State
0	dewitt	16798	IL
1	lacquiparle	8067	MN
2	lewisandclark	55716	MT
3	stjohnthebaptist	43044	LS

```
def canonicalize_county(county_name):
    return (county_name
    .lower() # lowercase
    .replace(' ', '')
    .replace('&', 'and') # remove spaces/dots, change & to and
    .replace('.', '')
    .replace('county', '')
    .replace('parish', '') # remove county/parish
    )
```

Example 6.2: Extraction

How do we extract specific days, etc. from a log file date?


```
pertinent = line.split("[")[1].split(']')[0] # get the date within brackets - get after the
↪ opening bracket and split off anything after closing bracket
day, month, rest = pertinent.split('/') # split by /
year, hour, minute, rest = rest.split(':') # split YYYY:HH:MM
seconds, time_zone = rest.split(' ') # split the rest
```

String Methods Reference

Operation	Python	Pandas Series
Transformation	`s.lower()/s.upper()`	`ser.str.lower()/ser.str.upper()`
Replacement/Deletion	`s.replace(<find>, <repl='')`	`ser.str.replace(<find>, <repl='')`
Split	`s.split(<keyword>)`	`ser.str.split(<keyword>)`
Substring	`s[1:4]`	`ser.str[1:4]`
Membership	`'ab' in s`	`ser.str.contains('ab')`
Length	`len(s)`	`ser.str.len()`

However, Python string functions are very brittle - it requires maintenance if the data changes, and it is also not as flexible (e.g. finding all the `moo+n` patterns in a string would be difficult). This often makes the extraction and canonicalization functions very "hacky" and messy.

An alternative is to use Regex, or regular expressions with the Python `re` library and the pandas `str` accessor.

Example 6.3: Extraction, Revisited

How can we change the code from the log file to a regular expression?

```
import re
pattern = r'\[(\d+)\/(\w+)\/(\d+):(\d+):(\d+):(\d+) (.+)\]'
day, month, year, hour, minute, seconds, time_zone = re.findall(pattern, line)[0]
```

6.2 Regex

Definition 6.1: Formal Language

Set of strings, typically described implicitly. (e.g. set of all strings with `len(s) < 10` with the phrase `'data'`)

Definition 6.2: Regular Language

Formal language that can be described by a regular expression.

Definition 6.3: Regular Expression

Sequence of characters that specifies a search pattern.

e.g. to describe SSNs, we can use the following sequence: `[0-9]\{3\}-[0-9]\{2\}[0-9]\{4\}` (3 of any digit, dash, 2 digits, dash, 4 digits)

Basic Regex Syntax

The four basic operators for Regex - you can do anything with these 4 operations.

Operation	Order	Example	Matches	Doesn't Match
Concatenation	3	`AABAAB`	`AABAAB`	every other string
Or	4	`AA|BAAB`	either `AA, BAAB`	every other string
Closure (zero or more)	2	`AB*A`	`AA, ABBBBBBA`	`AB, ABABA`
Group (parentheses)	1	`A(A|B)AAB, (AB)*A`	`A, ABABABABA`	`AA, ABBA`

Note 6.1

Note that `|, *, ()` are metacharacters, only manipulating adjacent characters.

e.g. `AB*` means one A, then 0 or more Bs, while `(AB)*` means 0 or more sets of AB

Example 6.4

Find a regular expression that matches "moon", "moooon" and so on (nonzero even # of o).

```
pattern = r"moo(oo)*n"
```

Follow up: find a regular expression that matches any nonzero even number of o or u.

```
pattern = r"m(oo|uu)(oo|uu)*n"
```

Expanded Regex Syntax

Operation	Example	Matches	Doesn't Match
Any character (except `\n`)	`.U.U.U.`	`CUMULUS, JUGULUM`	`SUCCUMBUS, TUMULTUOUS`
Character class	`[A-Za-z][a-z]*`	`word, Capitalized`	`camelCase, 4illegal`
Repeat `a` times	`j[aeiou]\{3\}hn`	`jaoehn, jooohn`	`jhn, jaeiouhn`
Repeat `\{a, b\}` times	`j[ou]\{1,2\}hn`	`john, juohn`	`jhn, jooohn`
At least 1	`jo+hn`	`john, jooohn`	`jhn, jjohn`
Zero or One	`joh?n`	`jon, john`	Any other string

Example 6.5

Find a regular expression for the following:

1. Any lowercase string with a repeated vowel (e.g. "noon", "peel", etc.).
2. String containing a lowercase letter and a number.

```
pattern1 = r"[a-z]*(aa|ee|ii|oo|uu)[a-z]*"
pattern2 = r"(.*[a-z].*[0-9].*)|(.*[0-9].*[a-z].*)"
```

Convenient Regex Syntax

Operation	Example	Matches	Doesn't Match
Built-in char classes	`\w+, \d+, \s+`	`Fawef_03, 12312, <whitespace>`	`a cup, 423 p, <non-ws>`
Char class negation (not in char class)	`[^a-z]+`	`PEPPERS191, 1711!`	`porch, Clams`
Escape Character (char literally)	`cow\.com`	`cow.com`	`cowscom`

Note: `\w = [A-Za-z0-9_], \d = [0-9], \s = <whitespace>`.

Example 6.6: Log File, Revisited

Give a regular expression to match the date portion in the log file.

```
pattern = r"\[.*\]"
```

Additional Regex Features

Operation	Example	Matches	Doesn't Match
Beginning of Line	`\^ark`	`ark two, ark o ark`	`dark`
End of Line	`ark\$`	`dark, ark o ark`	`ark two`
Lazy version of 0+/ `*`	`5.*?5`	`5005, 55`	`500505`

6.2.1 Regex in Python/Pandas

Canonicalization Python: We can use the `re.sub` function from the `re` module, replacing all instances of `pattern` with `repl`.

Example:

```
text = "<div><td valign="top">Moo</td></div>"
pattern = r"<[^>]+>" # html tags
re.sub(pattern, '', text) # replaces html tags with empty char, returns Moo
```

Pandas: Use `ser.str.replace(pattern, repl, regex=True)` to replace all instances of `pattern` with `repl`.

Example:

```
pattern = r"<[^>]+>"
df["Html"].str.replace(pattern, '', regex=True)
```

Note 6.2: Raw Strings

Use raw strings `r''/r""` as opposed to regular strings when using Regex. Both Regex and Python use backslash `\\` to escape characters, so it would lead to much more cluttered and uglier regular expressions (e.g. `"\\\\section"` vs `r"\\section"`)

Extraction

Python: Use `re.findall(pattern, text)` to find all instances of a pattern in the text.

Example:

```
text = "My social security number is 123-45-6789 bro, or actually maybe it's 321-45-6789.";
pattern = r"[0-9]{3}-[0-9]{2}-[0-9]{4}"
re.findall(pattern, text) # finds all valid numbers that could be SSNs
```

Pandas: `ser.str.findall(pattern)` returns a series of lists of valid matches.

Example:

```
pattern = r"[0-9]{3}-[0-9]{2}-[0-9]{4}"
df["SSN"].str.findall(pattern)
```

Capture Groups

Parentheses were used to specify the order of operations at first, but they also thought of as a match/capture group, which are returned as tuples of groups.

Example:

```
text = """Observations: 03:04:53 - Horse awakens.
03:05:14 - Horse goes back to sleep."""
pattern = "(\d\d):(\d\d):(\d\d) - (.*)"
matches = re.findall(pattern, text)
# returns [('03', '04', '53', 'Horse awakens.'), ('03', '05', '14', 'Horse goes back to sleep.')]
```

Note 6.3: Findall vs. Extract

Another function is `ser.str.extract(pattern)` and `ser.str.extractall(pattern)` which returns a dataframe of the first match (one group per column) / of all matches, one group per column, one row per match.

Note 6.4: Limitations for Regex

Writing regex is like writing a program: you need to know the syntax well, can be easier to write than read, and can be difficult to debug.

Regex is also not the best at some problems:

- Parsing hierarchical structures (e.g. JSON)
- Complex Features (e.g. valid email addresses)

- Counting (e.g. find same numbers of two characters, which is impossible in regex)
- Complex properties (e.g. palindromes, balanced parentheses, which are impossible in regex)

All in all, regex is still decent at wrangling text data.

6.3 Restaurant Data Demo