In [1]:
```
from dsc80_utils import *
```

Lecture 5 – Exploratory Data Analysis and Data Cleaning

DSC 80, Fall 2025

Agenda 📆

- Merging datasets.
- Dataset overview.
- Introduction to `plotly`.
- Exploratory data analysis and feature types.
- Data cleaning.
    - Data quality checks.
    - Missing values.
    - Transformations and timestamps.
    - Modifying structure.
- Investigating student-submitted questions!

Merging

Example: Name categories

The New York Times article from Lecture 1 claims that certain categories of names are becoming more popular. For example:

- Forbidden names like Lucifer, Lilith, Kali, and Danger.

- Evangelical names like Amen, Savior, Canaan, and Creed.

- Mythological names.

- It also claims that baby boomer names are becoming less popular.

Let's see if we can verify these claims using data!

Loading in the data

Our first DataFrame, `baby`, is the same as we saw in Lecture 1. It has one row for every combination of `'Name'`, `'Sex'`, and `'Year'`.

In [2]:
```
baby_path = Path('data') / 'baby.csv'
baby = pd.read_csv(baby_path)
baby
```

Out[2]:

	Name	Sex	Count	Year
0	Liam	M	20456	2022
1	Noah	M	18621	2022
2	Olivia	F	16573	2022
...	...	...	...	...
2085155	Wright	M	5	1880
2085156	York	M	5	1880
2085157	Zachariah	M	5	1880

2085158 rows × 4 columns

Our second DataFrame, `nyt`, contains the New York Times' categorization of each of several names, based on the aforementioned article.

In [3]:
```
nyt_path = Path('data') / 'nyt_names.csv'
nyt = pd.read_csv(nyt_path)
nyt
```

Out[3]:

	nyt_name	category
0	Lucifer	forbidden
1	Lilith	forbidden
2	Danger	forbidden
...	...	...
20	Venus	celestial
21	Celestia	celestial
22	Skye	celestial

23 rows × 2 columns

Issue: To find the number of babies born with (for example) forbidden names each year, we need to combine information from both `baby` and `nyt`.

Merging

- We want to link rows from `baby` and `nyt` together whenever the names match up.

- This is a **merge** (`pandas` term), i.e. a **join** (SQL term).
- A merge is appropriate when we have two sources of information **about the same individuals** that is **linked by a common column(s)**.
- The common column(s) are called the **join key**.

Example merge

Let's demonstrate on a small subset of `baby` and `nyt`.

In [4]:
```
nyt_small = nyt.iloc[[11, 12, 14]].reset_index(drop=True)

names_to_keep = ['Julius', 'Karen', 'Noah']
baby_small = (baby
 .query("Year == 2020 and Name in @names_to_keep")
 .reset_index(drop=True)
)

dfs_side_by_side(baby_small, nyt_small)
```

	Name	Sex	Count	Year
0	Noah	M	18407	2020
1	Julius	M	966	2020
2	Karen	F	330	2020
3	Noah	F	306	2020
4	Karen	M	6	2020

	nyt_name	category
0	Karen	boomer
1	Julius	mythology
2	Freya	mythology

In [5]:
```
baby_small.merge(nyt_small, left_on='Name', right_on='nyt_name')
```

Out[5]:

	Name	Sex	Count	Year	nyt_name	category
0	Julius	M	966	2020	Julius	mythology
1	Karen	F	330	2020	Karen	boomer
2	Karen	M	6	2020	Karen	boomer

The `merge` method

- The `merge` DataFrame method joins two DataFrames by columns or indexes.
  - As mentioned before, "merge" is just the `pandas` word for "join."
- When using the `merge` method, the DataFrame before `merge` is the "left" DataFrame, and the DataFrame passed into `merge` is the "right" DataFrame.

- In `baby_small.merge(nyt_small)`, `baby_small` is considered the "left" DataFrame and `nyt_small` is the "right" DataFrame; the columns from the left DataFrame appear to the left of the columns from right DataFrame.

- By default:
  - If join keys are not specified, all shared columns between the two DataFrames are used.
  - The "type" of join performed is an inner join. **This is the only type of join you saw in DSC 10, but there are more, as we'll now see!**

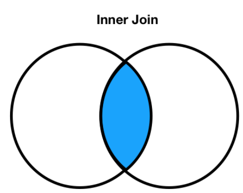
Join types: inner joins

In [6]:
```
baby_small.merge(nyt_small, left_on='Name', right_on='nyt_name')
```

Out[6]:

	Name	Sex	Count	Year	nyt_name	category
0	Julius	M	966	2020	Julius	mythology
1	Karen	F	330	2020	Karen	boomer
2	Karen	M	6	2020	Karen	boomer

- Note that `'Noah'` and `'Freya'` do not appear in the merged DataFrame.
- This is because there is:
  - no `'Noah'` in the right DataFrame (`nyt_small`), and
  - no `'Freya'` in the left DataFrame (`baby_small`).
- The default type of join that `merge` performs is an **inner join**, which keeps the **intersection** of the join keys.



Different join types

We can change the type of join performed by changing the `how` argument in `merge`. Let's experiment!

In [7]:
```
# Note the NaNs!
baby_small.merge(nyt_small, left_on='Name', right_on='nyt_name', how='left')
```

Out[7]:

	Name	Sex	Count	Year	nyt_name	category
0	Noah	M	18407	2020	NaN	NaN
1	Julius	M	966	2020	Julius	mythology
2	Karen	F	330	2020	Karen	boomer
3	Noah	F	306	2020	NaN	NaN
4	Karen	M	6	2020	Karen	boomer

In [8]:
```
baby_small.merge(nyt_small, left_on='Name', right_on='nyt_name', how='right'
```

Out[8]:

	Name	Sex	Count	Year	nyt_name	category
0	Karen	F	330.0	2020.0	Karen	boomer
1	Karen	M	6.0	2020.0	Karen	boomer
2	Julius	M	966.0	2020.0	Julius	mythology
3	NaN	NaN	NaN	NaN	Freya	mythology

In [9]:
```
baby_small.merge(nyt_small, left_on='Name', right_on='nyt_name', how='outer'
```

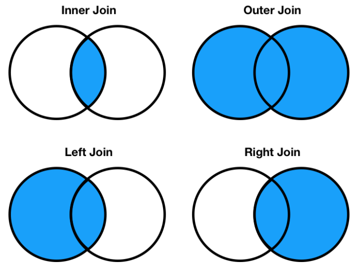
Out[9]:

	Name	Sex	Count	Year	nyt_name	category
0	NaN	NaN	NaN	NaN	Freya	mythology
1	Julius	M	966.0	2020.0	Julius	mythology
2	Karen	F	330.0	2020.0	Karen	boomer
3	Karen	M	6.0	2020.0	Karen	boomer
4	Noah	M	18407.0	2020.0	NaN	NaN
5	Noah	F	306.0	2020.0	NaN	NaN

Different join types handle mismatches differently

There are four types of joins.

- **Inner**: keep **only** matching keys (intersection).
- **Outer**: keep **all** keys in both DataFrames (union).
- **Left**: keep all keys in the left DataFrame, whether or not they are in the right DataFrame.
- **Right**: keep all keys in the right DataFrame, whether or not they are in the left DataFrame.
  - Note that `a.merge(b, how='left')` contains the same information as `b.merge(a, how='right')`, just in a different order.



Notes on the `merge` method

- `merge` is flexible – you can merge using a combination of columns, or the index of the DataFrame.
- If the two DataFrames have the same column names, `pandas` will add `_x` and `_y` to the duplicated column names to avoid having columns with the same name (change these the `suffixes` argument).
- There is, in fact, a `join` method, but it's actually a wrapper around `merge` with fewer options.
- **As always, the documentation is your friend!**

Lots of `pandas` operations do an implicit outer join!

- `pandas` will almost always try to match up index values using an outer join.
- It won't tell you that it's doing an outer join, it'll just throw `NaN` s in your result!

In [10]:
```
df1 = pd.DataFrame({'a': [1, 2, 3]}, index=['hello', 'dsc80', 'students'])
df2 = pd.DataFrame({'b': [10, 20, 30]}, index=['dsc80', 'is', 'awesome'])
dfs_side_by_side(df1, df2)
```

	a
hello	1
dsc80	2
students	3

	b
dsc80	10
is	20
awesome	30

In [11]:
```
df1['a'] + df2['b']
```

Out[11]:
```
awesome      NaN
dsc80       12.0
hello        NaN
is           NaN
students     NaN
dtype: float64
```

In [12]:
```
pd.concat([df1, df2])
```

Out[12]:

	a	b
hello	1.0	NaN
dsc80	2.0	NaN
students	3.0	NaN
dsc80	NaN	10.0
is	NaN	20.0
awesome	NaN	30.0

Many-to-one & many-to-many joins

One-to-one joins

- So far in this lecture, the joins we have worked with are called **one-to-one** joins.
- Neither the left DataFrame (`baby_small`) nor the right DataFrame (`nyt_small`) contained any duplicates in the join key.
- What if there are duplicated join keys, in one or both of the DataFrames we are merging?

In [13]:
```
# Run this cell to set up the next example.
profs = pd.DataFrame(
[['Sam', 'UCB', 5],
 ['Sam', 'UCSD', 5],
 ['Janine', 'UCSD', 8],
 ['Marina', 'UIC', 7],
 ['Justin', 'OSU', 5],
 ['Soohyun', 'UCSD', 2],
 ['Suraj', 'UCB', 2]],
    columns=['Name', 'School', 'Years']
)

schools = pd.DataFrame({
    'Abr': ['UCSD', 'UCLA', 'UCB', 'UIC'],
    'Full': ['University of California San Diego', 'University of Californi
})

programs = pd.DataFrame({
    'uni': ['UCSD', 'UCSD', 'UCSD', 'UCB', 'OSU', 'OSU'],
    'dept': ['Math', 'HDSI', 'COGS', 'CS', 'Math', 'CS'],
    'grad_students': [205, 54, 281, 439, 304, 193]
})
```

Many-to-one joins

- Many-to-one joins are joins where **one** of the DataFrames contains duplicate values in the join key.

- The resulting DataFrame will preserve those duplicate entries as appropriate.

In [14]:
```
dfs_side_by_side(profs, schools)
```

	Name	School	Years
0	Sam	UCB	5
1	Sam	UCSD	5
2	Janine	UCSD	8
3	Marina	UIC	7
4	Justin	OSU	5
5	Soohyun	UCSD	2
6	Suraj	UCB	2

	Abr	Full
0	UCSD	University of California San Diego
1	UCLA	University of California, Los Angeles
2	UCB	University of California, Berkeley
3	UIC	University of Illinois Chicago

Note that when merging `profs` and `schools`, the information from `schools` is duplicated.

- `'University of California, San Diego'` appears three times.
- `'University of California, Berkeley'` appears twice.

In [15]:
```
profs.merge(schools, left_on='School', right_on='Abr', how='left')
```

Out[15]:

	Name	School	Years	Abr	Full
0	Sam	UCB	5	UCB	University of California, Berkeley
1	Sam	UCSD	5	UCSD	University of California San Diego
2	Janine	UCSD	8	UCSD	University of California San Diego
3	Marina	UIC	7	UIC	University of Illinois Chicago
4	Justin	OSU	5	NaN	NaN
5	Soohyun	UCSD	2	UCSD	University of California San Diego
6	Suraj	UCB	2	UCB	University of California, Berkeley

Many-to-many joins

Many-to-many joins are joins where both DataFrames have duplicate values in the join key.

In [16]:
```
dfs_side_by_side(profs, programs)
```

	Name	School	Years
0	Sam	UCB	5
1	Sam	UCSD	5
2	Janine	UCSD	8
3	Marina	UIC	7
4	Justin	OSU	5
5	Soohyun	UCSD	2
6	Suraj	UCB	2

	uni	dept	grad_students
0	UCSD	Math	205
1	UCSD	HDSI	54
2	UCSD	COGS	281
3	UCB	CS	439
4	OSU	Math	304
5	OSU	CS	193

Before running the following cell, try predicting the number of rows in the output.

In [17]:
```
profs.merge(programs, left_on='School', right_on='uni')
```

Out[17]:

	Name	School	Years	uni	dept	grad_students
0	Sam	UCB	5	UCB	CS	439
1	Sam	UCSD	5	UCSD	Math	205
2	Sam	UCSD	5	UCSD	HDSI	54
...	...	...	...	...	...	...
10	Soohyun	UCSD	2	UCSD	HDSI	54
11	Soohyun	UCSD	2	UCSD	COGS	281
12	Suraj	UCB	2	UCB	CS	439

13 rows × 6 columns

- `merge` stitched together every UCSD row in `profs` with every UCSD row in `programs`.
- Since there were 3 UCSD rows in `profs` and 3 in `programs`, there are $3 \cdot 3 = 9$ UCSD rows in the output. The same applies for all other schools.

Question 🤔

Fill in the blank so that the last statement evaluates to `True`.

```
df = profs.merge(programs, left_on='School', right_on='uni')
df.shape[0] == (____).sum()
```

Don't use `merge` (or `join`) in your solution!

In [18]:
```
dfs_side_by_side(profs, programs)
```

	Name	School	Years
0	Sam	UCB	5
1	Sam	UCSD	5
2	Janine	UCSD	8
3	Marina	UIC	7
4	Justin	OSU	5
5	Soohyun	UCSD	2
6	Suraj	UCB	2

	uni	dept	grad_students
0	UCSD	Math	205
1	UCSD	HDSI	54
2	UCSD	COGS	281
3	UCB	CS	439
4	OSU	Math	304
5	OSU	CS	193

In [19]:
```
# Your code goes here.
```

Returning back to our original question

Let's find the popularity of baby name categories over time. To start, we'll define a DataFrame that has one row for every combination of `'category'` and `'Year'`.

In [20]:
```
cate_counts = (
    baby
    .merge(nyt, left_on='Name', right_on='nyt_name')
    .groupby(['category', 'Year'])
    ['Count']
    .sum()
    .reset_index()
)
cate_counts
```

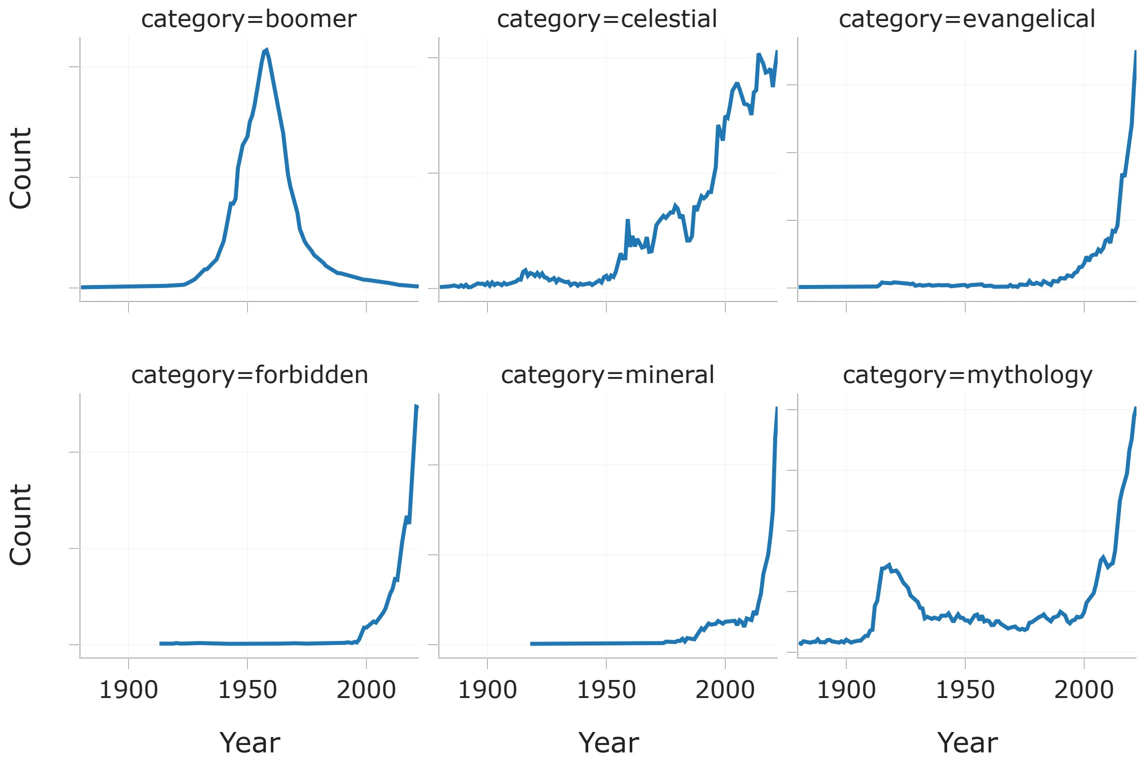
Out[20]:

	category	Year	Count
0	boomer	1880	292
1	boomer	1881	298
2	boomer	1882	326
...	...	...	...
659	mythology	2020	3516
660	mythology	2021	3895
661	mythology	2022	4049

662 rows × 3 columns

In [21]:
```
# We'll talk about plotting code soon!
import plotly.express as px
```

```
fig = px.line(cate_counts, x='Year', y='Count',
              facet_col='category', facet_col_wrap=3,
              facet_row_spacing=0.15,
              width=600, height=400)
fig.update_yaxes(matches=None, showticklabels=False)
fig
```



Transforming with `.apply`

Transforming values

- A **transformation** results from performing some operation on every element in a sequence, e.g. a Series.
- While we haven't discussed it yet in DSC 80, you learned how to transform Series in DSC 10, using the `apply` method. `apply` is very flexible – it takes in a function, which itself takes in a single value as input and returns a single value.

In [22]:
```
baby
```

Out[22]:

	Name	Sex	Count	Year
0	Liam	M	20456	2022
1	Noah	M	18621	2022
2	Olivia	F	16573	2022
...	...	...	...	...
2085155	Wright	M	5	1880
2085156	York	M	5	1880
2085157	Zachariah	M	5	1880

2085158 rows × 4 columns

In [23]:
```
def number_of_vowels(string):
    return sum(c in 'aeiou' for c in string.lower())

baby['Name'].apply(number_of_vowels)
```

Out[23]:
```
0          2
1          2
2          4
          ..
2085155    1
2085156    1
2085157    4
Name: Name, Length: 2085158, dtype: int64
```

In [24]:
```
# Built-in functions work with apply, too.
baby['Name'].apply(len)
```

Out[24]:
```
0          4
1          4
2          6
          ..
2085155    6
2085156    4
2085157    9
Name: Name, Length: 2085158, dtype: int64
```

The price of `apply`

Unfortunately, `apply` runs really slowly!

In [25]:
```
%%timeit
baby['Name'].apply(number_of_vowels)
```

```
1.1 s ± 22.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [ ]:
```
%%timeit
res = []
```

```
for name in baby['Name']:
    res.append(number_of_vowels(name))
```

Internally, `apply` actually just runs a `for`-loop in Python!

So, when possible – say, when applying arithmetic operations – we should work on Series objects directly and avoid `apply`!

The price of `apply`

In [ ]:
```
%%timeit
baby['Year'].apply(lambda y: y // 10 * 10)
```

In [ ]:
```
%%timeit
baby['Year'] // 10 * 10 # Rounds down to the nearest multiple of 10.
```

Apply is 100x slower!

The `.str` accessor

For string operations, `pandas` provides a convenient `.str` accessor.

In [ ]:
```
%%timeit
baby['Name'].str.upper()
```

In [ ]:
```
%%timeit
baby['Name'].apply(lambda s: s.upper())
```

It's very convenient and **runs about the same speed as `apply`!**

Dataset overview

San Diego food safety

From this article (archive link):

> In the last three years, one third of San Diego County restaurants have had at least one major food safety violation.

99% Of San Diego Restaurants Earn ‘A' Grades, Bringing Usefulness of System Into Question

From this article (archive link):

> Food held at unsafe temperatures. Employees not washing their hands. Dirty countertops. Vermin in the kitchen. An expired restaurant permit.
>
> Restaurant inspectors for San Diego County found these violations during a routine health inspection of a diner in La Mesa in November 2016. Despite the violations, the restaurant was awarded a score of 90 out of 100, the lowest possible score to achieve an 'A' grade.

The data

- We downloaded the data about the 1000 restaurants closest to UCSD from here.
- We had to download the data as JSON files, then process it into DataFrames. You'll learn how to do this soon!
  - Until now, you've (largely) been presented with CSV files that `pd.read_csv` could load without any issues.
  - But there are many different formats and possible issues when loading data in from files.
  - See Chapter 8 of Learning DS for more.

In [ ]:
```
rest_path = Path('data') / 'restaurants.csv'
insp_path = Path('data') / 'inspections.csv'
viol_path = Path('data') / 'violations.csv'
```

In [ ]:
```
rest = pd.read_csv(rest_path)
insp = pd.read_csv(insp_path)
viol = pd.read_csv(viol_path)
```

Question 🤔

The first article said that one third of restaurants had at least one major safety violation. Which DataFrames and columns seem most useful to verify this?

In [ ]:
```
rest.head(2)
```

In [ ]:
```
rest.columns
```

In [ ]:
```
insp.head(2)
```

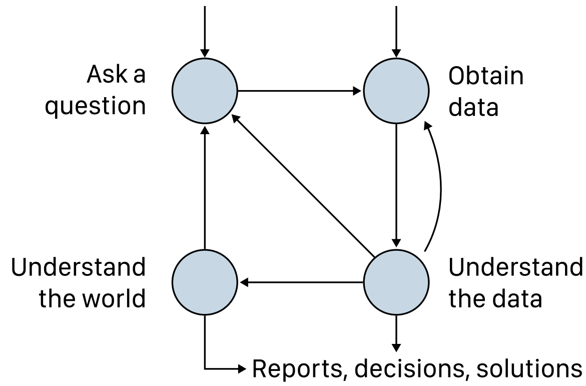
In [ ]:
```
insp.columns
```

In [ ]:
```
viol.head(2)
```

In [ ]:
```
viol.columns
```

Exploratory data analysis and feature types

The data science lifecycle, revisited



We're at the stage of **understanding the data**.

Exploratory data analysis (EDA)

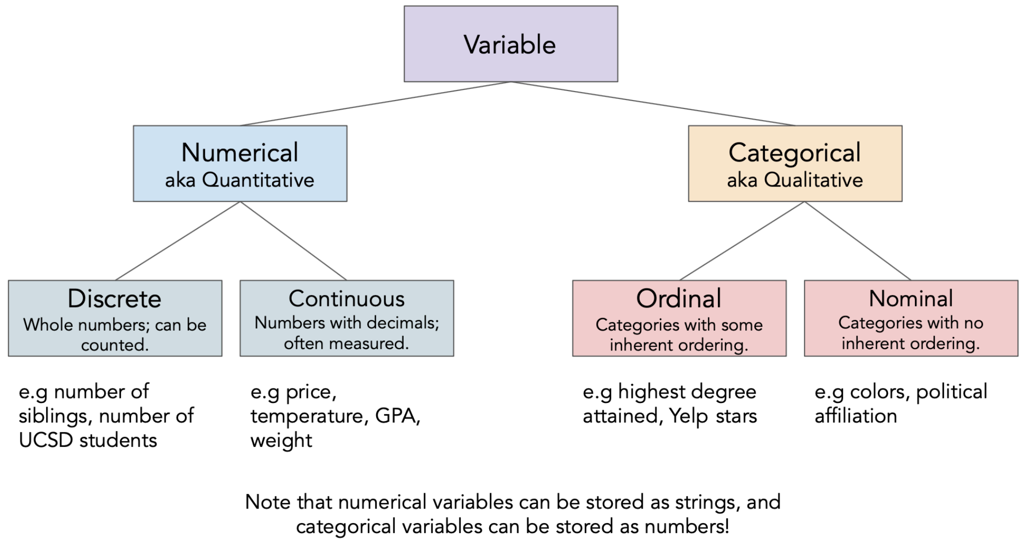
- Historically, data analysis was dominated by formal statistics, including tools like confidence intervals, hypothesis tests, and statistical modeling.
- In 1977, John Tukey defined the term **exploratory data analysis**, which described a philosophy for proceeding about data analysis:

  > Exploratory data analysis is actively incisive, rather than passively descriptive, with real emphasis on the discovery of the unexpected.

- Practically, EDA involves, among other things, computing summary statistics and drawing plots to understand the nature of the data at hand.

  > The greatest gains from data come from surprises… The unexpected is best brought to our attention by **pictures**.

Different feature types



Question 🤔

Determine the **feature type** of each of the following variables.

- `insp['score']`
- `insp['grade']`
- `viol['violation_accela']`
- `viol['major_violation']`
- `rest['business_id']`
- `rest['opened_date']`

In [ ]:
```
# Your code goes here.
```

Feature types vs. data types

- The data type `pandas` uses is not the same as the "data type" we talked about just now!
    - There's a difference between feature type (which has to do with the data's meaning) and computational data type (which has to with how it is stored).
- Take care when the two don't match up very well!

In [ ]:
```
# pandas stores these as ints, but they're actually nominal.
rest['business_id']
```

In [ ]:
```
# pandas stores these as strings, but they're actually numeric.
rest['opened_date']
```

Data cleaning

Four pillars of data cleaning

When loading in a dataset, to clean the data – that is, to prepare it for further analysis – we will:

1. Perform **data quality checks**.

2. Identify and handle **missing values**.

3. Perform **transformations**, including converting time series data to **timestamps**.

4. Modify **structure** as necessary.

Data cleaning: Data quality checks

Data quality checks

We often start an analysis by checking the quality of the data.

- Scope: Do the data match your understanding of the population?
- Measurements and values: Are the values reasonable?
- Relationships: Are related features in agreement?
- Analysis: Which features might be useful in a future analysis?

Scope

Do the data match your understanding of the population?

We were told that we're only looking at the 1000 restaurants closest to UCSD, so the restaurants in `rest` should agree with that.

In [ ]:
```
rest.sample(5)
```

Measurements and values

Are the values reasonable?

Do the values in the `'grade'` column match what we'd expect grades to look like?

In [ ]:
```
insp['grade'].value_counts()
```

What kinds of information does the `insp` DataFrame hold?

In [ ]:
```
insp.info()
```

What's going on in the `'address'` column of `rest` ?

In [ ]:
```
# Are there multiple restaurants with the same address?
rest['address'].value_counts()
```

In [ ]:
```
# Keeps all rows with duplicate addresses.
(
    rest
    .groupby('address')
    .filter(lambda df: df.shape[0] >= 2)
    .sort_values('address')
)
```

In [ ]:
```
# Does the same thing as above!
(
    rest[rest.duplicated(subset=['address'], keep=False)]
    .sort_values('address')
)
```

Relationships

Are related features in agreement?

Do the `'address'` es and `'zip'` codes in `rest` match?

In [ ]:
```
rest[['address', 'zip']]
```

What about the `'score'` s and `'grade'` s in `insp` ?

In [ ]:
```
insp[['score', 'grade']]
```

Analysis

Which features might be useful in a future analysis?

- We're most interested in:
  - These columns in the `rest` DataFrame: `'business_id'` , `'name'` , `'address'` , `'zip'` , and `'opened_date'` .
  - These columns in the `insp` DataFrame: `'business_id'` , `'inspection_id'` , `'score'` , `'grade'` , `'completed_date'` , and `'status'` .
  - These columns in the `viol` DataFrame: `'inspection_id'` , `'violation'` , `'major_violation'` , `'violation_text'` , and

`'violation_accela'`.

- Also, let's rename a few columns to make them easier to work with.

💡 Pro-Tip: Using `pipe`

When we manipulate DataFrames, it's best to define individual functions for each step, then use the `pipe` **method** to chain them all together.

The `pipe` DataFrame method takes in a function, which itself takes in a DataFrame and returns a DataFrame.

- In practice, we would add functions one by one to the top of a notebook, then `pipe` them all.
- For today, will keep re-running `pipe` to show data cleaning process.

In [ ]:
```
def subset_rest(rest):
    return rest[['business_id', 'name', 'address', 'zip', 'opened_date']]

rest = (
    pd.read_csv(rest_path)
    .pipe(subset_rest)
)
rest
```

In [ ]:
```
# Same as the above – but the above makes it easier to chain more .pipe call
subset_rest(pd.read_csv(rest_path))
```

Let's use `pipe` to keep (and rename) the subset of the columns we care about in the other two DataFrames as well.

In [ ]:
```
def subset_insp(insp):
    return (
        insp[['business_id', 'inspection_id', 'score', 'grade', 'completed_d
        .rename(columns={'completed_date': 'date'})
    )

insp = (
    pd.read_csv(insp_path)
    .pipe(subset_insp)
)
```

In [ ]:
```
def subset_viol(viol):
    return (
        viol[['inspection_id', 'violation', 'major_violation', 'violation_ac
        .rename(columns={'violation': 'kind',
                         'major_violation': 'is_major',
                         'violation_accela': 'violation'})
    )

viol = (
```

```
    pd.read_csv(viol_path)
    .pipe(subset_viol)
)
```

Combining the restaurant data

Let's join all three DataFrames together so that we have all the data in a single DataFrame.

In [ ]:
```
def merge_all_restaurant_data():
    return (
        rest
        .merge(insp, on='business_id', how='left')
        .merge(viol, on='inspection_id', how='left')
    )

df = merge_all_restaurant_data()
df
```

Question 🤔

Why should the function above use two left joins? What would go wrong if we used other kinds of joins?

We will probably end lecture here.

Data cleaning: Missing values

Missing values

Next, it's important to check for and handle missing values, as they can have a big effect on your analysis.

In [ ]:
```
insp[['score', 'grade']]
```

In [ ]:
```
# The proportion of values in each column that are missing.
insp.isna().mean()
```

In [ ]:
```
# Why are there null values here?
# insp['inspection_id'] and viol['inspection_id'] don't have any null values
df[df['inspection_id'].isna()]
```

There are many ways of handling missing values, which we'll cover in an entire lecture next week. But a good first step is to check how many there are!

Data cleaning: Transformations and timestamps

Transformations and timestamps

From last class:

> A transformation results from performing some operation on every element in a sequence, e.g. a Series.

It's often useful to look at ways of transforming your data to make it easier to work with.

- Type conversions (e.g. changing the string `"$2.99"` to the number `2.99`).
- Unit conversion (e.g. feet to meters).
- Extraction (Getting `'vermin'` out of `'Vermin Violation Recorded on 10/10/2023'`).

Creating timestamps

Most commonly, we'll parse dates into `pd.Timestamp` objects.

In [ ]:
```
# Look at the dtype!
insp['date']
```

In [ ]:
```
# This magical string tells Python what format the date is in.
# For more info: https://docs.python.org/3/library/datetime.html#strftime-ar
date_format = '%Y-%m-%d'
pd.to_datetime(insp['date'], format=date_format)
```

In [ ]:
```
# Another advantage of defining functions is that we can reuse this functior
# for the 'opened_date' column in `rest` if we wanted to.
def parse_dates(insp, col):
    date_format = '%Y-%m-%d'
    dates = pd.to_datetime(insp[col], format=date_format)
    return insp.assign(**{col: dates})

insp = (
    pd.read_csv(insp_path)
    .pipe(subset_insp)
    .pipe(parse_dates, 'date')
)

# We should also remake df, since it depends on insp.
```

```
# Note that the new insp is used to create df!
df = merge_all_restaurant_data()
```

In [ ]:
```
# Look at the dtype now!
df['date']
```

Working with timestamps

- We often want to adjust granularity of timestamps to see overall trends, or seasonality.
- Use the `resample` method in `pandas` (documentation).
  - Think of it like a version of `groupby`, but for timestamps.
  - For instance, `insp.resample('2W', on='date')` separates every two weeks of data into a different group.

In [ ]:
```
insp.resample('2W', on='date')['score'].mean()
```

In [ ]:
```
# Where are those numbers coming from?
insp[
    (insp['date'] >= pd.Timestamp('2020-01-05')) &
    (insp['date'] < pd.Timestamp('2020-01-19'))
]['score']
```

In [ ]:
```
(insp.resample('2W', on='date')
 .size()
 .plot(title='Number of Inspections Over Time')
)
```

The `.dt` accessor

Like with Series of strings, `pandas` has a `.dt` accessor for properties of timestamps (documentation).

In [ ]:
```
insp['date']
```

In [ ]:
```
insp['date'].dt.day
```

In [ ]:
```
insp['date'].dt.dayofweek
```

In [ ]:
```
dow_counts = insp['date'].dt.dayofweek.value_counts()
fig = px.bar(dow_counts)
fig.update_xaxes(tickvals=np.arange(7), ticktext=['Mon', 'Tues', 'Wed', 'Thu
```

Data cleaning: Modifying structure

Reshaping DataFrames

We often **reshape** the DataFrame's structure to make it more convenient for analysis. For example, we can:

- Simplify structure by removing columns or taking a set of rows for a particular period of time or geographic area.
  - We already did this!
- Adjust granularity by aggregating rows together.
  - To do this, use `groupby` (or `resample`, if working with timestamps).
- Reshape structure, most commonly by using the DataFrame `melt` method to un-pivot a dataframe.

Using `melt`

- The `melt` method is common enough that we'll give it a special mention.
- We'll often encounter pivot tables (esp. from government data), which we call *wide* data.
- The methods we've introduced work better with *long-form* data, or *tidy* data.
- To go from wide to long, `melt`.

Wide-form:

	Jan	Feb	Mar
2001	10	20	30
2002	130	200	340

Long-form:

Year	Month	Seats
2001	Jan	10
2001	Feb	20
2001	Mar	30
2002	Jan	130
2002	Feb	200
2002	Mar	340

Example usage of `melt`

In [ ]:
```
wide_example = pd.DataFrame({
    'Year': [2001, 2002],
    'Jan': [10, 130],
    'Feb': [20, 200],
    'Mar': [30, 340]
}).set_index('Year')
wide_example
```

In [ ]:
```
wide_example.melt(ignore_index=False)
```

Exploration

Question 🤔 (Answer at dsc80.com/q)

Code: `qs`

What questions do you want me to try and answer with the data? I'll start with a single pre-prepared question, and then answer student questions until we run out of time.

Example question: Can we rank restaurants by their number of violations? How about separately for each zip code?

And why would we want to do that? 🤔

```
In [ ]:
```

Summary, next time

Summary

- Data cleaning is a necessary starting step in data analysis. There are four pillars of data cleaning:
  - Quality checks.
  - Missing values.
  - Transformations and timestamps.
  - Modifying structure.
- Approach EDA with an open mind, and draw lots of visualizations.

Next time

Hypothesis and permutation testing. Some of this will be DSC 10 review, but we'll also push further!