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

Lecture 6 – EDA Part 2, Hypothesis Testing

DSC 80, Fall 2025

In case you need a review from DSC 10, there is a **Pre-Lecture Review** for this lecture.

Agenda 📆

- Finish up our EDA example from last lecture
- Data scope.
- Overview of hypothesis testing.
- Example: Total variation distance.
- Permutation testing.
  - Example: Birth weight and smoking 🚬.
  - Example (that you'll read on your own): Permutation testing meets TVD.

SD Food Safety Data

First, let's retrace our steps from last lecture.

In [2]:
```
rest_path = Path('data') / 'restaurants.csv'
insp_path = Path('data') / 'inspections.csv'
viol_path = Path('data') / 'violations.csv'

rest = pd.read_csv(rest_path)
insp = pd.read_csv(insp_path)
viol = pd.read_csv(viol_path)
```

Like with most real world data, this needs some cleaning.

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**.

Loading [MathJax]/extensions/Safe.jsand 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 [3]:
```
rest.sample(5)
```

Loading [MathJax]/extensions/Safe.js

Out[3]:

	business_id	name	business_type	address	...	lat	long
536	211995356480	GROM GRUB	Limited Food Prep Cart	9932 MESA RIM RD, SAN DIEGO, CA 92121-3930	...	32.90	-117.18
61	211900216875	CANTEEN PACIRA 10450	Pre-Packaged Retail Market	10450 SCIENCE CENTER DR, SAN DIEGO, CA 92121-1119	...	32.89	-117.23
968	211898486874	CIRCLE K #5095	Retail Market with Deli	4360 GENESEE AVE, SAN DIEGO, CA 92117-4901	...	32.82	-117.18
839	211899386954	CADMAN ELEMENTARY	School Processing Food Facility	4370 KAMLOOP AVE, SAN DIEGO, CA 92117	...	32.82	-117.22
473	211922896691	INSTRUMENTATION LABS	Pre-Packaged Retail Market	6260 SEQUENCE DR, SAN DIEGO, CA 92121-4358	...	32.90	-117.19

5 rows × 12 columns

Measurements and values

Are the values reasonable?

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

In [4]:
```
insp['grade'].value_counts()
```

Out[4]:
```
grade
A    2978
B      11
Name: count, dtype: int64
```

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

Loading [MathJax]/extensions/Safe.js

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5179 entries, 0 to 5178
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   custom_id       5179 non-null   object 
 1   business_id     5179 non-null   int64  
 2   inspection_id   5179 non-null   int64  
 3   description     0 non-null      float64
 4   type            5179 non-null   object 
 5   score           5179 non-null   int64  
 6   grade           2989 non-null   object 
 7   completed_date  5179 non-null   object 
 8   status          5179 non-null   object 
 9   link            5179 non-null   object 
 10  status_link     5179 non-null   object 
dtypes: float64(1), int64(3), object(7)
memory usage: 445.2+ KB
```

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

In [6]:
```
# Are there multiple restaurants with the same address?
rest['address'].value_counts()
```

Out[6]:
```
address
5300 GRAND DEL MAR CT, SAN DIEGO, CA 92130       9
8657 VILLA LA JOLLA DR, LA JOLLA, CA 92037       8
4545 LA JOLLA VILLAGE DR, SAN DIEGO, CA 92122    8
                                                ..
3963 GOVERNOR DR, SAN DIEGO, CA 92122            1
4041 GOVERNOR DR, SAN DIEGO, CA 92122-2520       1
2672 DEL MAR HEIGHTS RD, DEL MAR, CA 92014       1
Name: count, Length: 863, dtype: int64
```

In [7]:
```
# let's look at the rows with duplicate addresses.
(
    rest
    .groupby('address')
    .filter(lambda df: df.shape[0] >= 2)
    .sort_values('address')
)
```

Loading [MathJax]/extensions/Safe.js

Out[7]:

	business_id	name	business_type	address	...	lat	long	opened_
406	211899308875	NASEEMS BAKERY & KABOB	Restaurant Food Facility	10066 PACIFIC HEIGHTS BLVD, SAN DIEGO, CA 92121	...	32.90	-117.19	2012-(
402	211898699154	HANAYA SUSHI CAFE	Restaurant Food Facility	10066 PACIFIC HEIGHTS BLVD, SAN DIEGO, CA 92121	...	32.90	-117.19	2011-(
401	211899558107	ARMANDOS MEXICAN FOOD	Restaurant Food Facility	10066 PACIFIC HEIGHTS BLVD, SAN DIEGO, CA 92121	...	32.90	-117.19	2005-(
...	...	...	...	...	...	...	...	
575	211972411855	TARA HEATHER CAKE DESIGN	Caterer	9932 MESA RIM RD, SUITE# A, SAN DIEGO, CA 9212...	...	32.90	-117.18	2014-(
344	211990537315	COMPASS GROUP FEDEX EXPRESS OLSON	Pre-Packaged Retail Market	9999 OLSON DR, SAN DIEGO, CA 92121-2837	...	32.89	-117.20	2022-
343	211976587262	CANTEEN - FED EX OLSON	Pre-Packaged Retail Market	9999 OLSON DR, SAN DIEGO, CA 92121-2837	...	32.89	-117.20	2020-(

213 rows × 12 columns

Loading [MathJax]/extensions/Safe.js

Relationships

Are related features in agreement?

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

In [8]:
```
rest[['address', 'zip']]
```

Loading [MathJax]/extensions/Safe.js

Out[8]:

	address	zip
0	3233 LA JOLLA VILLAGE DR, LA JOLLA, CA 92037	92037
1	8950 VILLA LA JOLLA DR, SUITE# B123, LA JOLLA,...	92037-1704
2	6902 LA JOLLA BLVD, LA JOLLA, CA 92037	92037
...	...	...
997	1234 TOURMALINE ST, SAN DIEGO, CA 92109-1856	92109-1856
998	12925 EL CAMINO REAL, SUITE# AA4, SAN DIEGO, C...	92130
999	2672 DEL MAR HEIGHTS RD, DEL MAR, CA 92014	92014

1000 rows × 2 columns

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

In [9]:
```
insp[['score', 'grade']]
```

Out[9]:

	score	grade
0	96	NaN
1	98	NaN
2	98	NaN
...	...	...
5176	0	NaN
5177	0	NaN
5178	90	A

5179 rows × 2 columns

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'`.

Loading [MathJax]/extensions/Safe.js

- 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 [10]:
```
def subset_rest(rest):
    return rest[['business_id', 'name', 'address', 'zip', 'opened_date']]

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

Out[10]:

	business_id	name	address	zip	opened_date
0	211898487641	MOBIL MART LA JOLLA VILLAGE	3233 LA JOLLA VILLAGE DR, LA JOLLA, CA 92037	92037	2002-05-05
1	211930769329	CAFE 477	8950 VILLA LA JOLLA DR, SUITE# B123, LA JOLLA,...	92037-1704	2023-07-24
2	211909057778	VALLEY FARM MARKET	6902 LA JOLLA BLVD, LA JOLLA, CA 92037	92037	2019-01-22
...	...	...	...	...	...
997	211899338714	PACIFIC BEACH ELEMENTARY	1234 TOURMALINE ST, SAN DIEGO, CA 92109-1856	92109-1856	2002-05-05
998	211942150255	POKEWAN DEL MAR	12925 EL CAMINO REAL, SUITE# AA4, SAN DIEGO, C...	92130	2016-11-03
999	211925713322	SAFFRONO LOUNGE RESTAURANT	2672 DEL MAR HEIGHTS RD, DEL MAR, CA 92014	92014	2022-11-03

1000 rows × 5 columns

Loading [MathJax]/extensions/Safe.js

In [11]:
```
# Same as the above — but the above makes it easier to chain more .pipe call
subset_rest(pd.read_csv(rest_path))
```

Out[11]:

	business_id	name	address	zip	opened_date
0	211898487641	MOBIL MART LA JOLLA VILLAGE	3233 LA JOLLA VILLAGE DR, LA JOLLA, CA 92037	92037	2002-05-05
1	211930769329	CAFE 477	8950 VILLA LA JOLLA DR, SUITE# B123, LA JOLLA,...	92037-1704	2023-07-24
2	211909057778	VALLEY FARM MARKET	6902 LA JOLLA BLVD, LA JOLLA, CA 92037	92037	2019-01-22
...	...	...	...	...	...
997	211899338714	PACIFIC BEACH ELEMENTARY	1234 TOURMALINE ST, SAN DIEGO, CA 92109-1856	92109-1856	2002-05-05
998	211942150255	POKEWAN DEL MAR	12925 EL CAMINO REAL, SUITE# AA4, SAN DIEGO, C...	92130	2016-11-03
999	211925713322	SAFFRONO LOUNGE RESTAURANT	2672 DEL MAR HEIGHTS RD, DEL MAR, CA 92014	92014	2022-11-03

1000 rows × 5 columns

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

In [12]:
```
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 [13]:
```
def subset_viol(viol):
    return (
        viol[['inspection_id', 'violation', 'major_violation', 'violation_ac
        .rename(columns={'violation': 'kind',
                         'major_violation': 'is_major',
                         'violation_accela': 'violation'})
    )
```

Loading [MathJax]/extensions/Safe.js

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

In [14]:
```
def subset_rest(rest):
    return rest[['business_id', 'name', 'address', 'zip', 'opened_date']]

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

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

rest = (pd.read_csv(rest_path)
        .pipe(subset_rest))
insp = (pd.read_csv(insp_path)
        .pipe(subset_insp))
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 [15]:
```
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.head(2)
```

Loading [MathJax]/extensions/Safe.js

Out[15]:

	business_id	name	address	zip	...	status	kind	is_major	violation
0	211898487641	MOBIL MART LA JOLLA VILLAGE	3233 LA JOLLA VILLAGE DR, LA JOLLA, CA 92037	92037	...	Complete	Hot and Cold Water	Y	21. Hot & cold water available
1	211898487641	MOBIL MART LA JOLLA VILLAGE	3233 LA JOLLA VILLAGE DR, LA JOLLA, CA 92037	92037	...	Complete	Hot and Cold Water	N	21. Hot & cold water available

2 rows × 13 columns

Question 🤔

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

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.

`.info()` quickly tells you how many non-null values are in each column:

In [16]:
```
insp.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5179 entries, 0 to 5178
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   business_id    5179 non-null   int64 
 1   inspection_id  5179 non-null   int64 
 2   score          5179 non-null   int64 
 3   grade          2989 non-null   object
 4   date           5179 non-null   object
 5   status         5179 non-null   object
dtypes: int64(3), object(3)
e: 242.9+ KB
```

Loading [MathJax]/extensions/Safe.js

It looks like the `grade` column has a lot of nulls:

In [17]:
```
insp[['score', 'grade']]
```

Out[17]:

	score	grade
0	96	NaN
1	98	NaN
2	98	NaN
...	...	...
5176	0	NaN
5177	0	NaN
5178	90	A

5179 rows × 2 columns

In [18]:
```
# The proportion of values in each column that are missing.
insp.isna().mean()
```

Out[18]:
```
business_id      0.00
inspection_id    0.00
score            0.00
grade            0.42
date             0.00
status           0.00
dtype: float64
```

In [19]:
```
# Why are there null values here?
# insp['inspection_id'] and viol['inspection_id'] don't have any null values
df[df['inspection_id'].isna()]
```

Loading [MathJax]/extensions/Safe.js

Out[19]:

	business_id	name	address	zip	...	status	kind	is_major	vic
759	211941133403	TASTY CHAI	8878 REGENTS RD 105, SAN DIEGO, CA 92122-5853	92122-5853	...	NaN	NaN	NaN	
1498	211915545446	EMBASSY SUITES SAN DIEGO LA JOLLA	4550 LA JOLLA VILLAGE DR, SAN DIEGO, CA 92122-...	92122-1248	...	NaN	NaN	NaN	
1672	211937443689	SERVICENOW	4770 EASTGATE MALL, SAN DIEGO, CA 92121-1970	92121-1970	...	NaN	NaN	NaN	
...	...	...	...	...	...	...	...	...	
8094	211997340975	COOKIE SCOOP	7759 GASTON DR, SAN DIEGO, CA 92126-3036	92126-3036	...	NaN	NaN	NaN	
8450	211900595220	I LOVE BANANA BREAD CO	4068 DALLES AVE, SAN DIEGO, CA 92117-5518	92117-5518	...	NaN	NaN	NaN	
8545	211963768842	PETRA KITCHEN	5252 BALBOA ARMS DR 175, SAN DIEGO, CA 92117-4949	92117-4949	...	NaN	NaN	NaN	

29 rows × 13 columns

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!

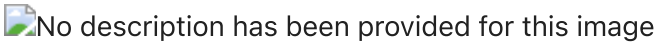
Quick Introduction to `plotly`

Loading [MathJax]/extensions/Safe.js

`plotly`

- We've used `plotly` in lecture briefly, and you even have to use it in Project 1 Question 13, but we haven't yet discussed it formally.

- It's a visualization library that enables **interactive** visualizations.



Using `plotly`

There are a few ways we can use `plotly`:

- Using the `plotly.express` syntax.
  - `plotly` is very flexible, but it can be verbose; `plotly.express` allows us to make plots quickly.
  - See the **documentation here** – it's very rich (there are good examples for almost everything).

- By setting `pandas` plotting backend to `'plotly'` (by default, it's `'matplotlib'`) and using the DataFrame `plot` method.
  - The DataFrame `plot` method is how you created plots in DSC 10!

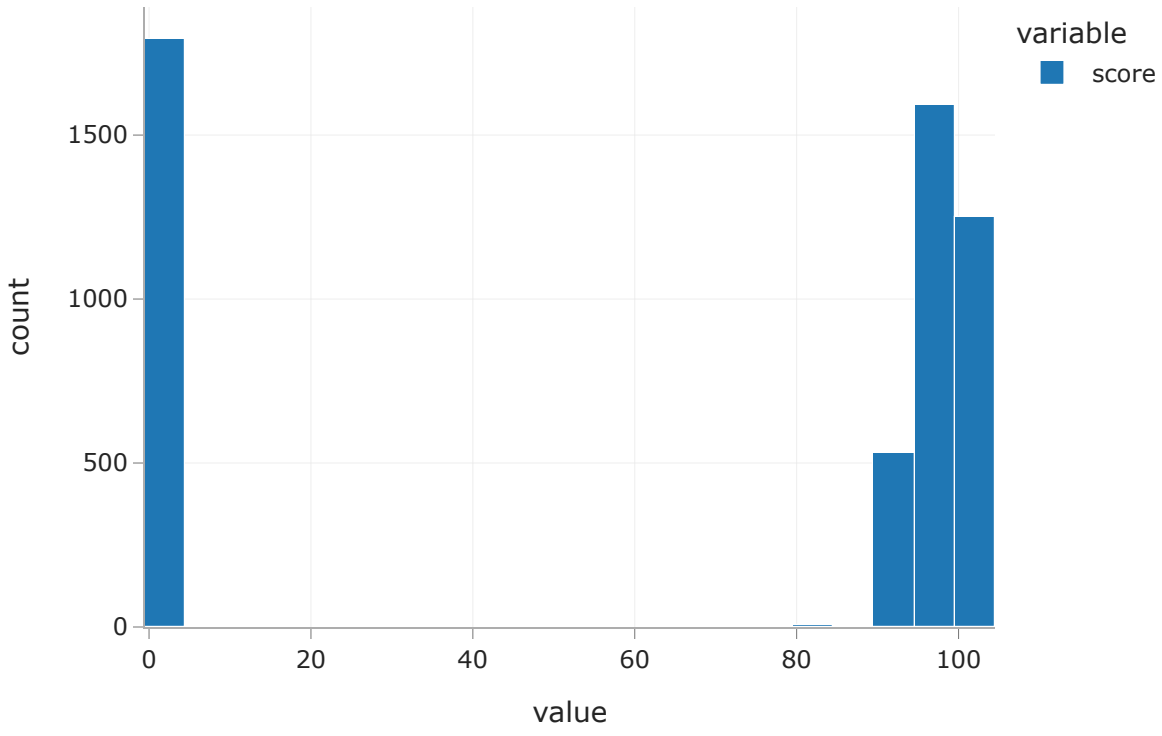
For now, we'll use `plotly.express` syntax; we've imported it in the `dsc80_utils.py` file that we import at the top of each lecture notebook.

Initial plots

First, let's look at the distribution of inspection `'score'`s:

In [20]:
```
fig = px.histogram(insp['score'])
fig
```

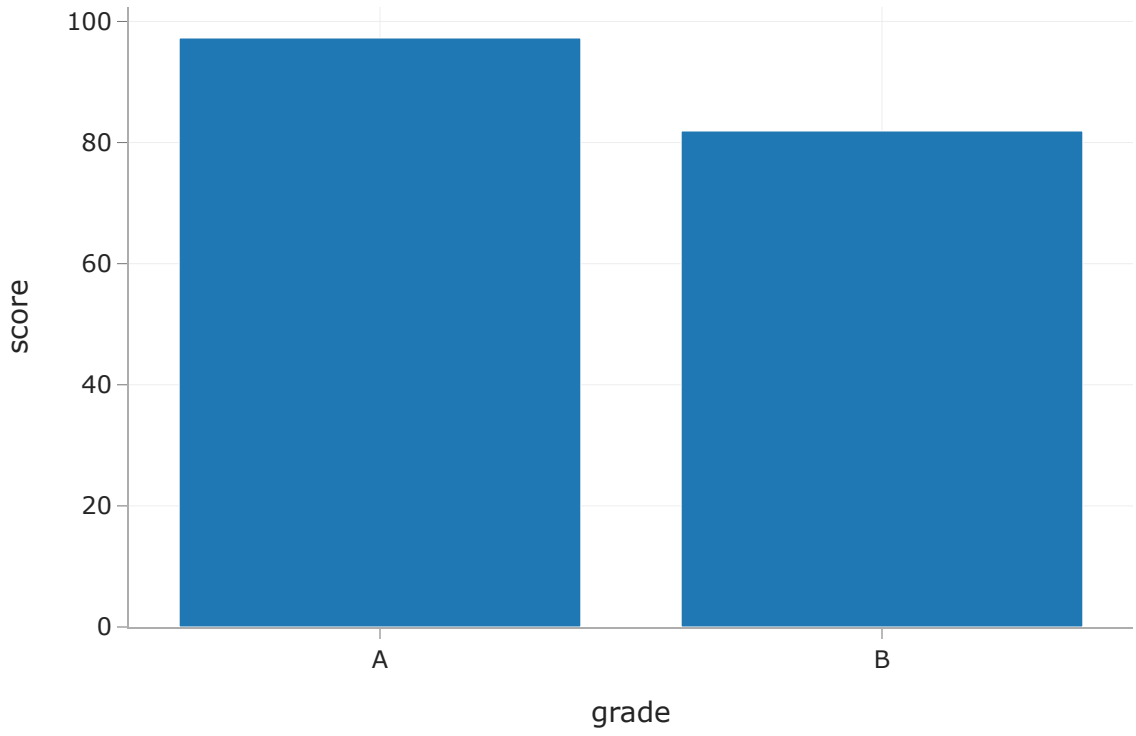
Loading [MathJax]/extensions/Safe.js



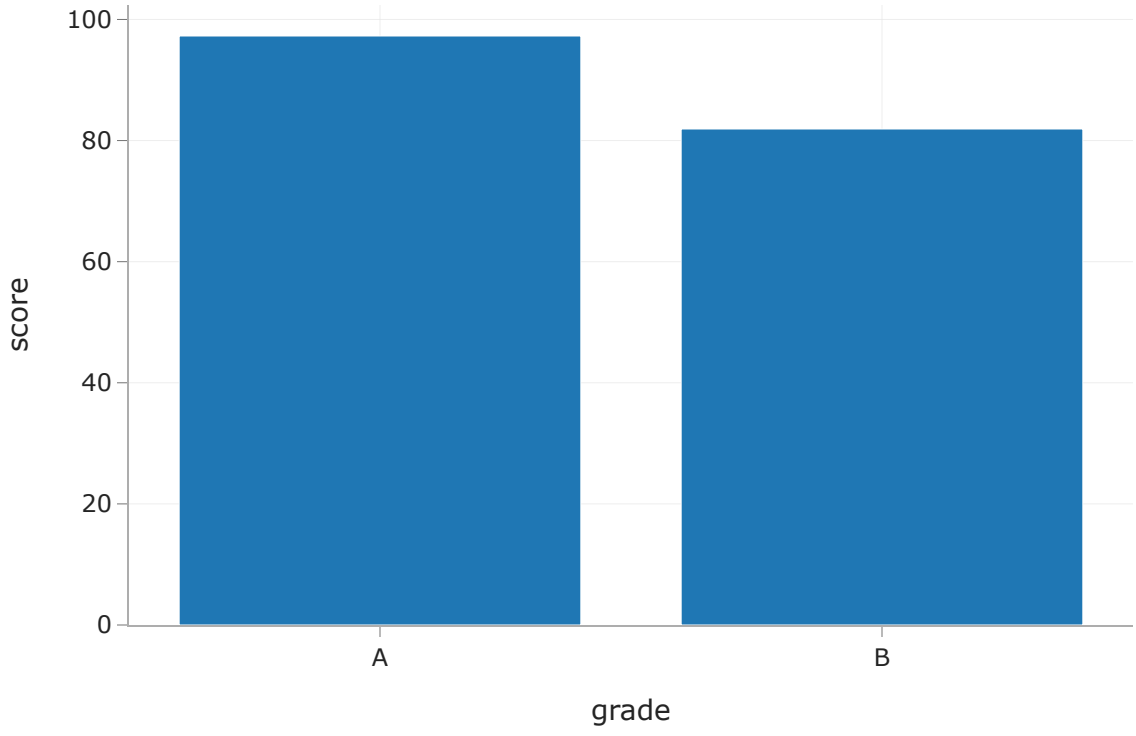
How about the distribution of average inspection `'score'` per `'grade'`?

In [21]:
```
scores = (
    insp[['grade', 'score']]
    .dropna()
    .groupby('grade')
    .mean()
    .reset_index()
)
# x= and y= are columns of scores. Convenient!
px.bar(scores, x='grade', y='score')
```

Loading [MathJax]/extensions/Safe.js



In [22]:
```
# Same as the above!
scores.plot(kind='bar', x='grade', y='score')
```



Data cleaning: Transformations and timestamps

Loading [MathJax]/extensions/Safe.js

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 [23]:
```
# Look at the dtype!
insp['date']
```

Out[23]:
```
0       2023-02-16
1       2022-01-03
2       2020-12-03
           ...
5176    2023-03-06
5177    2022-12-09
5178    2022-11-30
Name: date, Length: 5179, dtype: object
```

Each date is being represented as a string. This is not very convenient. Instead, it's usually desirable to convert them to `pd.Timestamp` objects.

In [24]:
```
# 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)
```

Out[24]:
```
0      2023-02-16
1      2022-01-03
2      2020-12-03
          ...
5176   2023-03-06
5177   2022-12-09
5178   2022-11-30
Name: date, Length: 5179, dtype: datetime64[ns]
```

Loading [MathJax]/extensions/Safe.js

If you don't specify a format, `pd.to_datetime` tried to infer it (and usually does a good job):

In [25]:
```
pd.to_datetime(insp['date'])
```

Out[25]:
```
0      2023-02-16
1      2022-01-03
2      2020-12-03
          ...
5176   2023-03-06
5177   2022-12-09
5178   2022-11-30
Name: date, Length: 5179, dtype: datetime64[ns]
```

In [26]:
```
# 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 [27]:
```
# Look at the dtype now!
df['date']
```

Out[27]:
```
0      2023-02-16
1      2022-01-03
2      2020-12-03
          ...
8728   2022-11-30
8729   2022-11-30
8730   2022-11-30
Name: date, Length: 8731, dtype: datetime64[ns]
```

`pd.Timestamp` objects are nice

You can easily compare dates:

In [28]:
```
# find all inspections that happened before Oct. 5, 2022.
df['date'] < pd.Timestamp('2022-10-5')
```

Loading [MathJax]/extensions/Safe.js

Out[28]:
```
0       False
1        True
2        True
        ...
8728    False
8729    False
8730    False
Name: date, Length: 8731, dtype: bool
```

In [29]:
```
# find the date of the first inspection
df['date'].min()
```

Out[29]:
```
Timestamp('2020-01-02 00:00:00')
```

Can also do date arithmetic using `pd.Timedelta`

In [30]:
```
df['date']
```

Out[30]:
```
0      2023-02-16
1      2022-01-03
2      2020-12-03
          ...
8728   2022-11-30
8729   2022-11-30
8730   2022-11-30
Name: date, Length: 8731, dtype: datetime64[ns]
```

In [31]:
```
df['date'] + pd.Timedelta('3 days')
```

Out[31]:
```
0      2023-02-19
1      2022-01-06
2      2020-12-06
          ...
8728   2022-12-03
8729   2022-12-03
8730   2022-12-03
Name: date, Length: 8731, dtype: datetime64[ns]
```

Example: find all inspections that occurred within 30 days of the restaurant opening.

In [32]:
```
df['date'] < (pd.to_datetime(df['opened_date']) + pd.Timedelta('30 days'))
```

Out[32]:
```
0       False
1       False
2       False
        ...
8728     True
8729     True
8730     True
Length: 8731, dtype: bool
```

Example: how far apart were the first and last inspections in the data set?

```
.max() - df['date'].min()
```

Loading [MathJax]/extensions/Safe.js

Out[33]:
```
Timedelta('1380 days 00:00:00')
```

The `.dt` accessor

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

In [34]:
```
insp['date'].dt.year
```

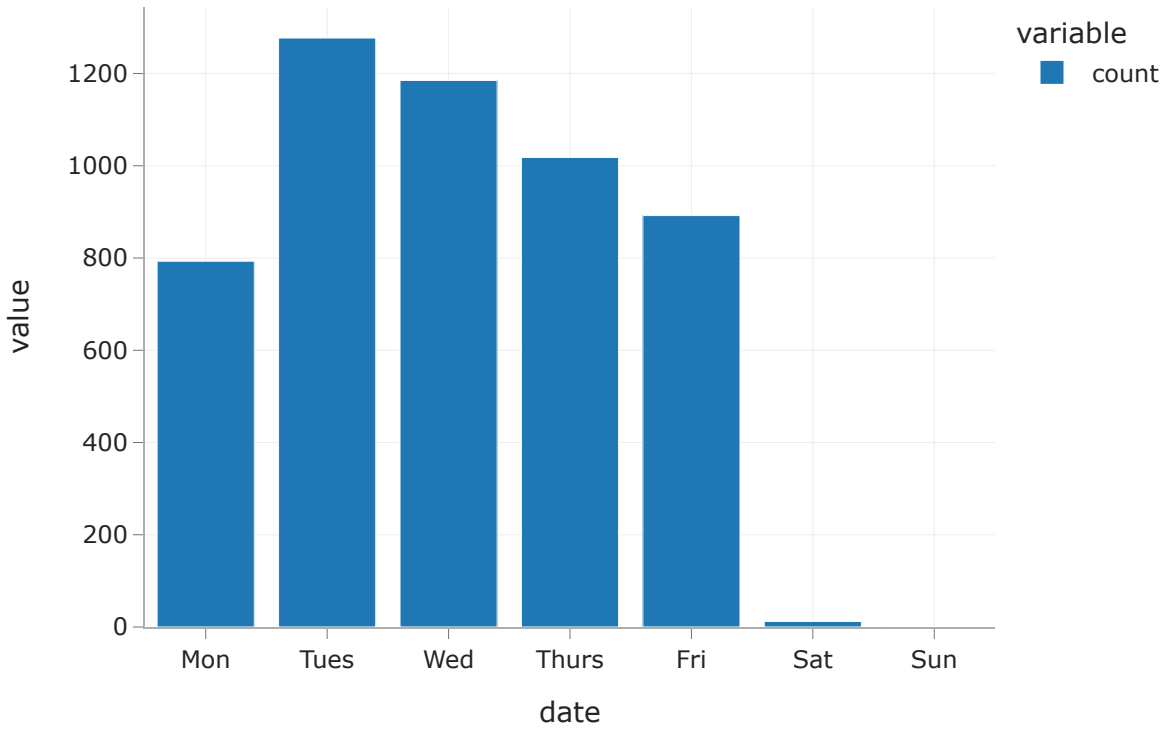
Out[34]:
```
0       2023
1       2022
2       2020
        ...
5176    2023
5177    2022
5178    2022
Name: date, Length: 5179, dtype: int32
```

In [35]:
```
insp['date'].dt.dayofweek
```

Out[35]:
```
0       3
1       0
2       3
       ..
5176    0
5177    4
5178    2
Name: date, Length: 5179, dtype: int32
```

In [36]:
```
(
    insp['date'].dt.dayofweek.value_counts()
    .plot.bar().update_xaxes(
        tickvals=np.arange(7), ticktext=['Mon', 'Tues', 'Wed', 'Thurs', 'Fri
    )
)
```

Loading [MathJax]/extensions/Safe.js



Adjusting granularity (grouping by time spans)

- 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.

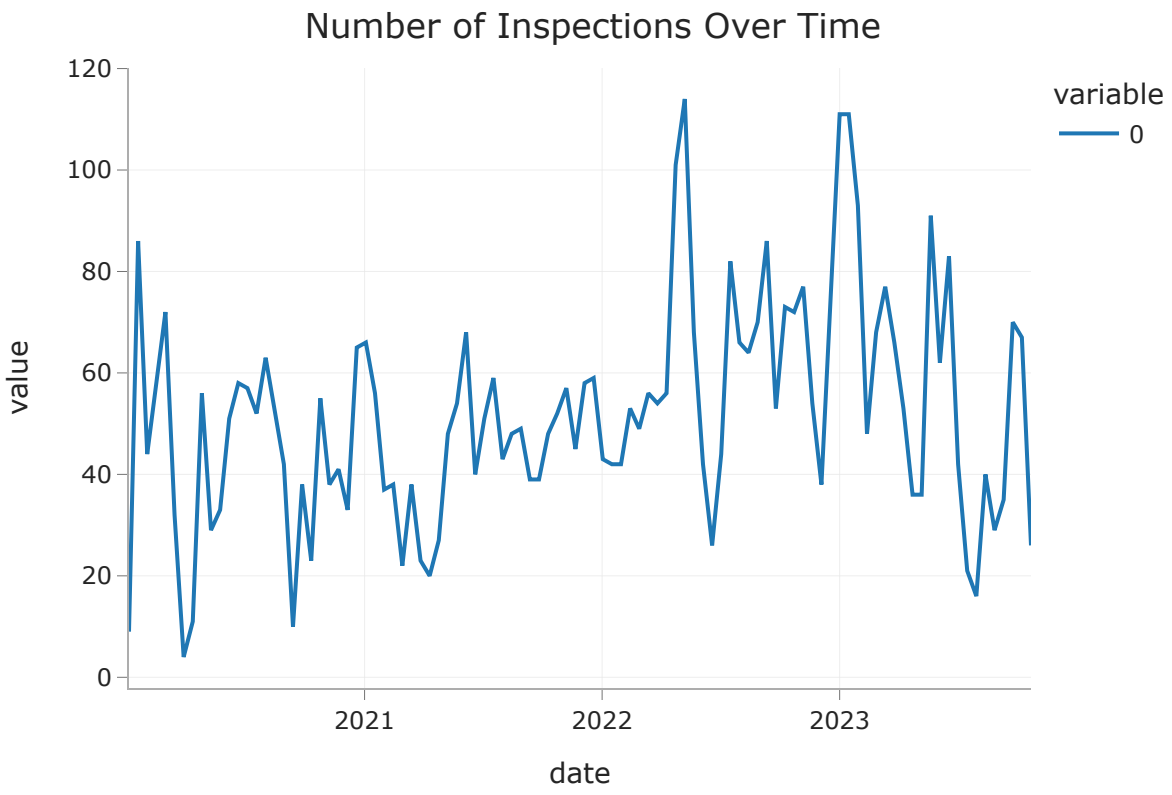
In [37]:
```
# we'll get one group for every 2 weeks of data:
# insp.resample('2W', on='date')['score'].apply(print)
```

In [38]:
```
insp.resample('2W', on='date')['score'].mean()
```

Out[38]:
```
date
2020-01-05    42.67
2020-01-19    59.33
2020-02-02    56.34
              ...
2023-09-24    66.60
2023-10-08    59.58
2023-10-22    66.81
Freq: 2W-SUN, Name: score, Length: 100, dtype: float64
```

In [39]:
```
(insp.resample('2W', on='date')
 .size()
 .plot(title='Number of Inspections Over Time')
)
```

Loading [MathJax]/extensions/Safe.js



Exploration

Question 🤔

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 [40]:
```
# your code here
```

Hypothesis Testing

Why are we learning hypothesis testing again?

You may say,

> Didn't we already learn this in DSC 10?

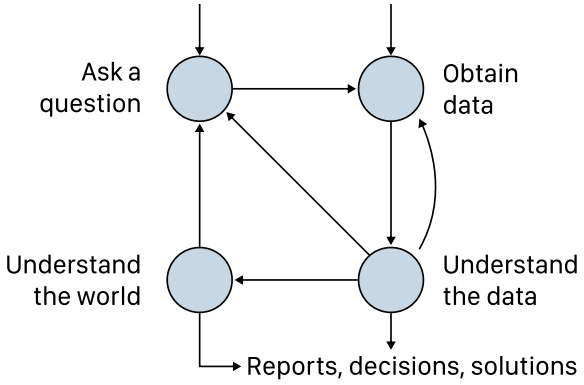
Yes, but:

Loading [MathJax]/extensions/Safe.js

- It's an important concept, but one that's confusing to understand the first time you learn about it.
- In addition, in order to properly handle missing values (next lecture), we need to learn how to identify different **missingness mechanisms**. Doing so requires performing a hypothesis test.

Data scope

Where are we in the data science lifecycle?



Hypothesis testing is a tool for helping us **understand the world (some population)**, given our **understanding of the data (some sample)**.

Data scope

- **Statistical inference**: The practice of drawing conclusions about a population, given a sample.
- **Target population**: All elements of the population you ultimately want to draw conclusions about.
- **Access frame**: All elements that are accessible for you for measurement and observation.
- **Sample**: The subset of the access frame that you actually measured / observed.

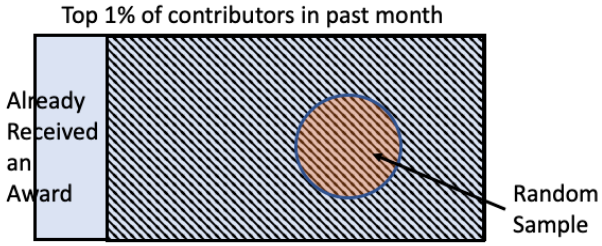
Example: Wikipedia awards

A 2012 paper asked:

> If we give awards to Wikipedia contributors, will they contribute more?

To test this question, they took the top 1% of all Wikipedia contributors, excluded those who already received an award, and then took a random sample of 200 contributors.

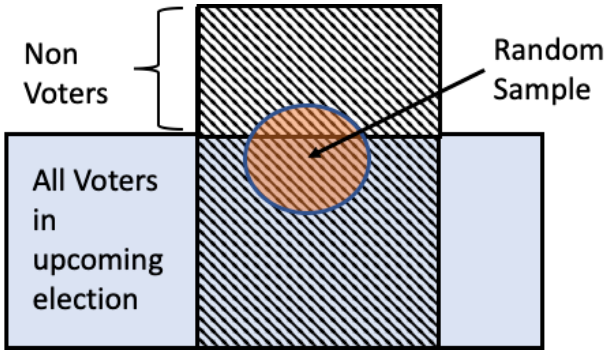
Loading [MathJax]/extensions/Safe.js



Example: Who will win the election?

In the 2016 US Presidental Election, most pollsters predicted Clinton to win over Trump, even though Trump ultimately won.

To poll, they randomly selected **potential** voters and asked them a question over the phone.



🔑 Key Idea: Random samples look like the access frame they were sampled from!

- This enables statistical inference!
- But keep in mind, random samples look like their access frame which can be different than the population itself.

Sampling in practice

In DSC 10, you used a few key functions/methods to draw samples from populations.

- To draw samples from a known sequence (e.g. array or Series), you used `np.random.choice`.

In [41]:
```
names = np.load(Path('data') / 'names.npy', allow_pickle=True)

# By default, the sampling is done WITH replacement.
np.random.choice(names, 10)
```

Loading [MathJax]/extensions/Safe.js

Out[41]:
```
array(['Aritra', 'Kailey', 'Aile', 'Brendan', 'Katelyn', 'Zening', 'Jack',
       'Eshaan', 'Tiffany', 'Tatiana'], dtype=object)
```

In [42]:
```
# To sample WITHOUT replacement, set replace=False.
# This is known as "simple random sampling."
np.random.choice(names, 10, replace=False)
```

Out[42]:
```
array(['Hannah', 'Lacha', 'Samuel', 'Yujin', 'Yash', 'Allen', 'Yihui',
       'Brandon', 'Krystal', 'Tianqi'], dtype=object)
```

- The DataFrame `.sample` method also allowed you to draw samples from a known sequence.

In [43]:
```
# Samples WITHOUT replacement by default (the opposite of np.random.choice).
pd.DataFrame(names, columns=['name']).sample(10)
```

Out[43]:

	name
71	Nadine
91	Subika
105	Yiran
...	...
82	Rushil
5	Amelia
103	Yeogyeong

10 rows × 1 columns

- To sample from a **categorical** distribution, you used `np.random.multinomial`. Note that in the cell below, we don't see `array([50, 50])` every time, and that's due to randomness!

In [44]:
```
# Draws 100 elements from a population in which 50% are group 0 and 50% are
# This sampling is done WITH replacement.
# In other words, each sampled element has a 50% chance of being group 0 and
np.random.multinomial(100, [0.5, 0.5])
```

Out[44]:
```
array([37, 63])
```

Overview of hypothesis testing

What problem does hypothesis testing solve?

Suppose we've performed an experiment, or identified something interesting in our data.

Loading [MathJax]/extensions/Safe.js

- Say we've created a new vaccine.
- To assess its efficiency, we give one group the vaccine, and another a placebo.
- We notice that the flu rate among those who received the vaccine is lower than among those who received the placebo (i.e. didn't receive the vaccine).
- One possibility: the vaccine doesn't actually do anything, and by chance, those with the vaccine happened to have a lower flu rate.
- Another possibility: receiving the vaccine made a difference – the flu rate among those who received the vaccine is lower than we'd expect due to random chance.
- **Hypothesis testing allows us to determine whether an observation is "significant."**

Why hypothesis testing is difficult to learn

- It's like "[proof by contradiction](https://brilliant.org/wiki/contradiction/#:~:text=Proof%20by%20contradiction%20(also
- If I want to show that my vaccine works, I consider a world where it doesn't (null hypothesis).
- Then, I show that under the null hypothesis my data would be very unlikely.
- Why go through these mental hurdles? Showing something is not true is usually easier than showing something is true!

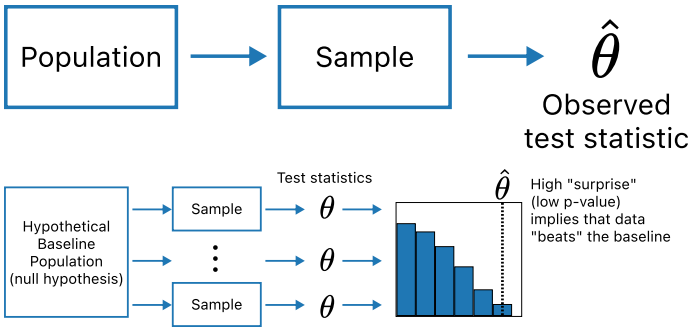
The hypothesis testing "recipe"

Faced with a question about the data raised by an observation...

1. Decide on null and alternate hypotheses.
    - The null hypothesis should be a well-defined probability model that reflects the baseline you want to compare against.
    - The alternative hypothesis should be the "alternate reality" that you suspect may be true.
2. Decide on a **test statistic**, such that a large observed statistic would point to one hypothesis and a small observed statistic would point to the other.
3. Compute an empirical distribution of the test statistic under the null by drawing samples from the null hypothesis' probability model.

Loading [MathJax]/extensions/Safe.js

4. Assess whether the observed test statistic is consistent with the empirical distribution of the test statistic by computing a **p-value**.



Example: Total variation distance

Ethnic distribution of California vs. UCSD

The DataFrame below contains the ethnic breakdown of the state as a whole (source) and UCSD as of 2016 (source).

In [45]:
```
eth = pd.DataFrame(
    [['Asian', 0.15, 0.51],
     ['Black', 0.05, 0.02],
     ['Latino', 0.39, 0.16],
     ['White', 0.35, 0.2],
     ['Other', 0.06, 0.11]],
    columns=['Ethnicity', 'California', 'UCSD']
).set_index('Ethnicity')

eth
```

Out[45]:

Ethnicity	California	UCSD
Asian	0.15	0.51
Black	0.05	0.02
Latino	0.39	0.16
White	0.35	0.20
Other	0.06	0.11

- The two distributions above are clearly different.
- One possibility: UCSD students **do** look like a random sample of California residents, and the distributions above look different purely due to random chance.

Loading [MathJax]/extensions/Safe.js

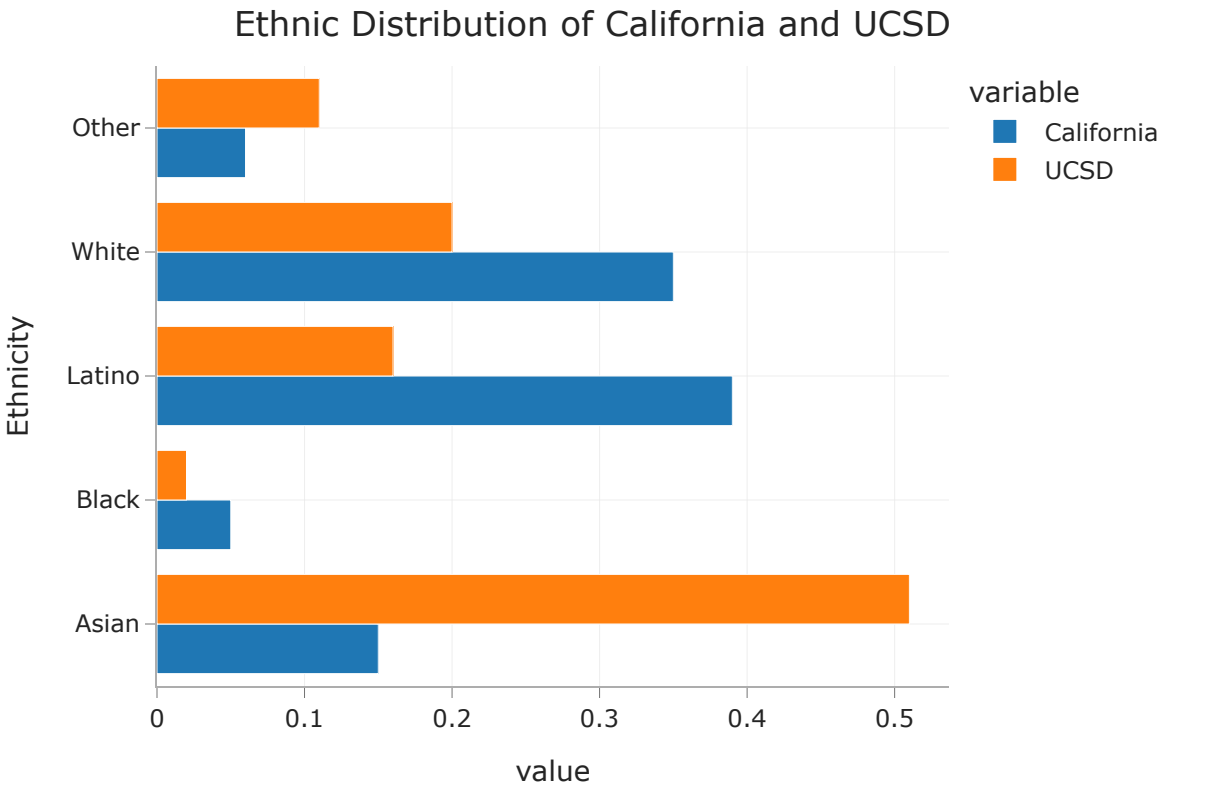
- Another possibility: UCSD students **don't** look like a random sample of California residents, because the distributions above look too different.

Is the difference between the two distributions significant?

Let's establish our hypotheses.

- **Null Hypothesis**: UCSD students **were** selected at random from the population of California residents.
- **Alternative Hypothesis**: UCSD students **were not** selected at random from the population of California residents.
- **Observation**: Ethnic distribution of UCSD students.
- **Test Statistic**: We need a way of quantifying **how different** two categorical distributions are.

In [46]:
```
eth.plot(kind='barh', title='Ethnic Distribution of California and UCSD', ba
```



How can we summarize the difference, or **distance**, between these two distributions using just a single number?

Total variation distance

The total variation distance (TVD) is a test statistic that describes the **distance between two categorical distributions**.

Loading [MathJax]/extensions/Safe.js

If $A = [a_1, a_2, ..., a_k]$ and $B = [b_1, b_2, ..., b_k]$ are both categorical distributions, then the TVD between A and B is

$$\text{TVD}(A, B) = \frac{1}{2} \sum_{i = 1}^k \big|a_i - b_i\big|$$

Let's compute the TVD between UCSD's ethnic distribution and California's ethnic distribution. We *could* define a function to do this (and you can use this in assignments):

```
def tvd(dist1, dist2):
    return np.abs(dist1 - dist2).sum() / 2
```

But let's try and work on the `eth` DataFrame directly, using the `diff` method.

In [47]:
```
# The diff method finds the differences of consecutive elements in a Series.
pd.Series([4, 5, -2]).diff()
```

Out[47]:
```
0    NaN
1    1.0
2   -7.0
dtype: float64
```

In [48]:
```
observed_tvd = eth.diff(axis=1).abs().sum().iloc[1] / 2
observed_tvd
```

Out[48]:
```
np.float64(0.41000000000000003)
```

The issue is we don't know whether this is a large value or a small value – we don't know where it lies in the **distribution of TVDs under the null**.

The plan

To conduct our hypothesis test, we will:

- Repeatedly generate samples of size 30,000 (the number of UCSD students) from the ethnic distribution of all of California.
- Each time, compute the TVD between the simulated distribution and California's distribution.
- **This will generate an empirical distribution of TVDs, under the null.**
- Finally, determine whether the observed TVD (0.41) is consistent with the empirical distribution of TVDs.

Generating one random sample

Again, to sample from a categorical distribution, we use `np.random.multinomial`.

Important: We must sample from the "population" distribution here, which is the ethnic of everyone in California.

Loading [MathJax]/extensions/Safe.js

In [49]:
```
# Number of students at UCSD in this example.
N_STUDENTS = 30_000
```

In [50]:
```
eth['California']
```

Out[50]:
```
Ethnicity
Asian     0.15
Black     0.05
Latino    0.39
White     0.35
Other     0.06
Name: California, dtype: float64
```

In [51]:
```
np.random.multinomial(N_STUDENTS, eth['California'])
```

Out[51]:
```
array([ 4447,  1520, 11700, 10537,  1796])
```

In [52]:
```
np.random.multinomial(N_STUDENTS, eth['California']) / N_STUDENTS
```

Out[52]:
```
array([0.15, 0.05, 0.39, 0.35, 0.06])
```

Generating many random samples and computing TVDs, without a `for` -loop

We *could* write a `for` -loop to repeat the process on the previous slide repeatedly (and you *can* in labs and projects). However, the Pre-Lecture Review told us about the `size` argument in `np.random.multinomial` , so let's use that here.

In [53]:
```
eth_draws = np.random.multinomial(N_STUDENTS, eth['California'], size=100_00
eth_draws
```

Out[53]:
```
array([[0.15, 0.05, 0.39, 0.35, 0.06],
       [0.15, 0.05, 0.39, 0.35, 0.06],
       [0.15, 0.05, 0.38, 0.35, 0.06],
       ...,
       [0.15, 0.05, 0.38, 0.35, 0.06],
       [0.15, 0.05, 0.39, 0.35, 0.06],
       [0.15, 0.05, 0.39, 0.35, 0.06]])
```

In [54]:
```
eth_draws.shape
```

Out[54]:
```
(100000, 5)
```

Notice that each row of `eth_draws` sums to 1, because each row is a simulated categorical distribution.

In [55]:
```
# The values here appear rounded.
tvds = np.abs(eth_draws - eth['California'].to_numpy()).sum(axis=1) / 2
tvds
```

Loading [MathJax]/extensions/Safe.js

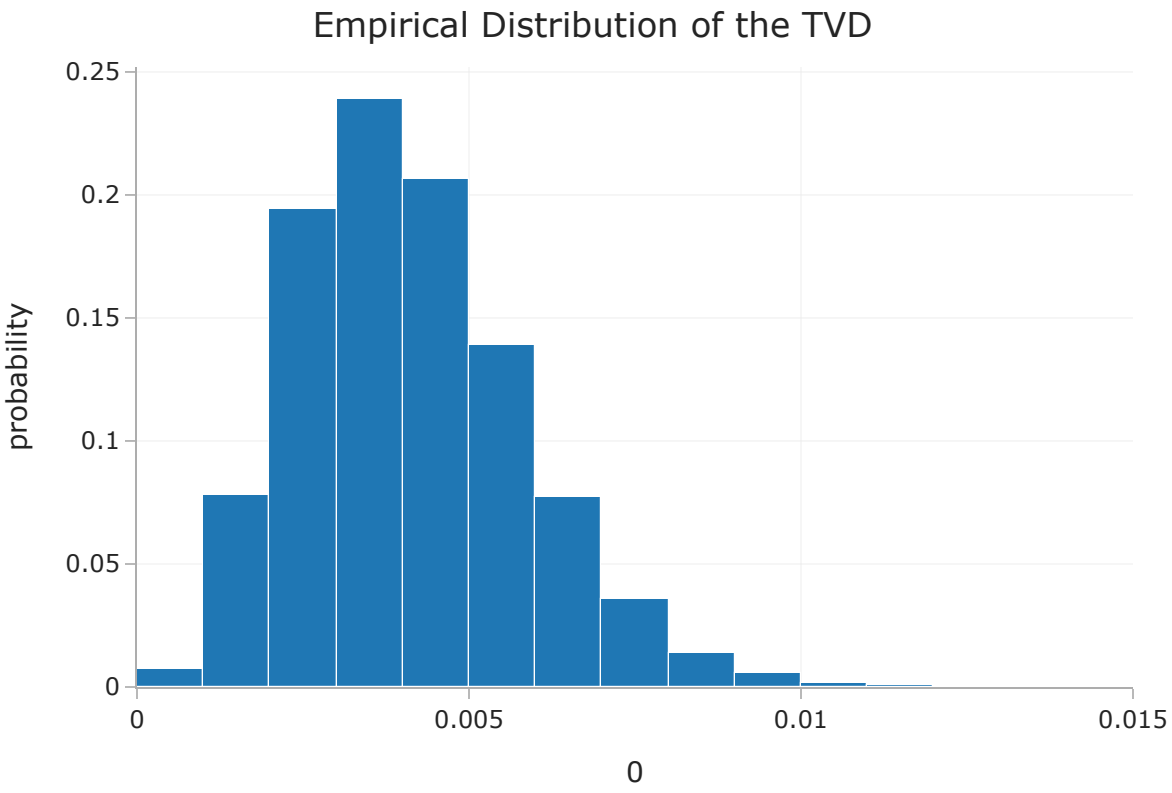
```
1, 0.  , 0.01, ..., 0.01, 0.  , 0.  ])
```

Visualizing the empirical distribution of the test statistic

In [56]:
```
observed_tvd
```

Out[56]:
```
np.float64(0.41000000000000003)
```

In [57]:
```
fig = px.histogram(pd.DataFrame(tvds), x=0, nbins=20, histnorm='probability'
                   title='Empirical Distribution of the TVD')
fig
```



In [58]:
```
(np.array(tvds) >= observed_tvd).mean()
```

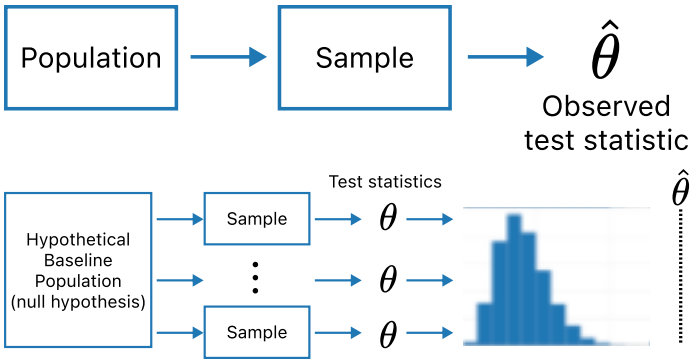
Out[58]:
```
np.float64(0.0)
```

No, there's not a mistake in our code!

Conclusion

- The chance that the observed TVD came from the distribution of TVDs under the null is essentially 0.
- This matches our intuition from the start – the two distributions looked very different to begin with. But now we're quite sure the difference can't be explained solely due to sampling variation.

Loading [MathJax]/extensions/Safe.js



Summary of the method

To assess whether an "observed sample" was drawn randomly from a known categorical distribution:

- Use the TVD as the test statistic because it measures the distance between two categorical distributions.
- Sample at random from the population. Compute the TVD between each random sample and the known distribution to get an idea for what reasonable deviations from the eligible pool look like. Repeat this process many, many times.
- Compare:
  - the empirical distribution of TVDs, with
  - the observed TVD from the sample.

Aside

- It was probably obvious that the difference is significant even before running a hypothesis test.
- Why? There are 30,000 students. Such a difference in proportion is unlikely to be due to random chance – there's something more systematic at play.
- But what if `N_STUDENTS = 300`, `N_STUDENTS = 30`, or `N_STUDENTS=3`?

In [59…
```
eth
```

Loading [MathJax]/extensions/Safe.js

Out[59]:

Ethnicity	California	UCSD
Asian	0.15	0.51
Black	0.05	0.02
Latino	0.39	0.16
White	0.35	0.20
Other	0.06	0.11

In [60…
```
def ethnicity_test(N_STUDENTS):
    eth_draws = np.random.multinomial(N_STUDENTS, eth['California'], size=1
    tvds = np.sum(np.abs(eth_draws - eth['California'].to_numpy()), axis=1
    return (np.array(tvds) >= observed_tvd).mean()
```

In [61…
```
for i in range(1, 9):
    N_STUDENTS = 2 ** i
    print(f'With {N_STUDENTS} students, the p-value is {ethnicity_test(N_S
```

```
With 2 students, the p-value is 0.72781.
With 4 students, the p-value is 0.30626.
With 8 students, the p-value is 0.08278.
With 16 students, the p-value is 0.00351.
With 32 students, the p-value is 1e-05.
With 64 students, the p-value is 0.0.
With 128 students, the p-value is 0.0.
With 256 students, the p-value is 0.0.
```

Permutation testing

Hypothesis testing vs. permutation testing

- So far, we've used hypothesis tests to answer questions of the form:

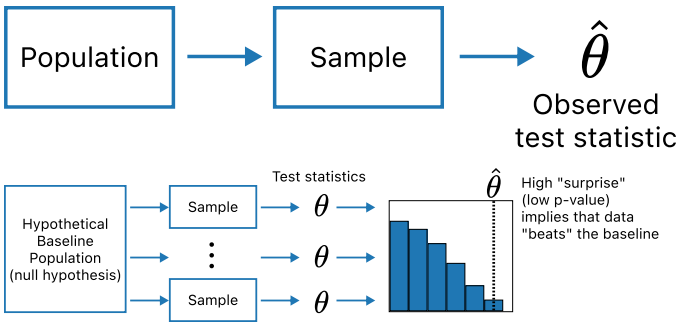
> I know the population distribution, and I have one sample. Is this sample a likely draw from the population?

- Next, we want to consider questions of the form:

> I have two samples, but no information about any population distributions. Do these samples look like they were drawn from different populations? That is, do these two samples look "different"?

Hypothesis testing vs. permutation testing

Loading [MathJax]/extensions/Safe.js



This framework requires us to be able to draw samples from the baseline population – but what if we don't know that population?

Example: Birth weight and smoking 🚬

For familiarity, we'll start with an example from DSC 10. This means we'll move quickly!

Let's start by loading in the data. Each row corresponds to a mother/baby pair.

In [62…
```
baby = pd.read_csv(Path('data') / 'babyweights.csv')
baby
```

Out[62]:

	Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Maternal Smoker
0	120	284	27	62	100	False
1	113	282	33	64	135	False
2	128	279	28	64	115	True
...	...	...	...	...	...	...
1171	130	291	30	65	150	True
1172	125	281	21	65	110	False
1173	117	297	38	65	129	False

1174 rows × 6 columns

We're only interested in the `'Birth Weight'` and `'Maternal Smoker'` columns.

In [63…
```
baby = baby[['Maternal Smoker', 'Birth Weight']]
baby.head()
```

Loading [MathJax]/extensions/Safe.js

Out[63]:

	Maternal Smoker	Birth Weight
0	False	120
1	False	113
2	True	128
3	True	108
4	False	136

Note that there are **two samples**:

- Birth weights of smokers' babies (rows where `'Maternal Smoker'` is `True`).
- Birth weights of non-smokers' babies (rows where `'Maternal Smoker'` is `False`).

Exploratory data analysis

How many babies are in each group? What is the average birth weight within each group?

In [64…
```
baby.groupby('Maternal Smoker')['Birth Weight'].agg(['mean', 'count'])
```

Out[64]:

	mean	count
Maternal Smoker		
False	123.09	715
True	113.82	459

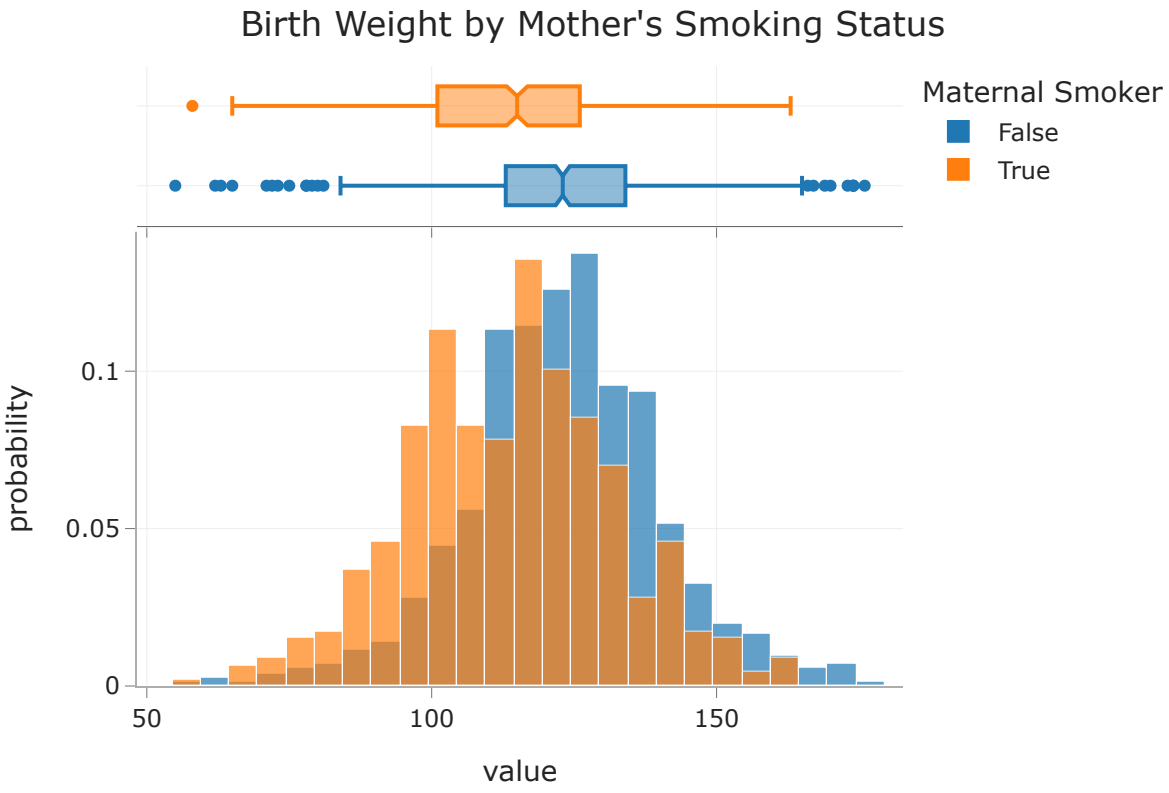
Note that 16 ounces are in 1 pound, so the above weights are ~7-8 pounds.

Visualizing birth weight distributions

Below, we draw the distributions of both sets of birth weights.

In [65…
```
fig = px.histogram(baby, color='Maternal Smoker', histnorm='probability', r
                   title="Birth Weight by Mother's Smoking Status", barmode
fig
```

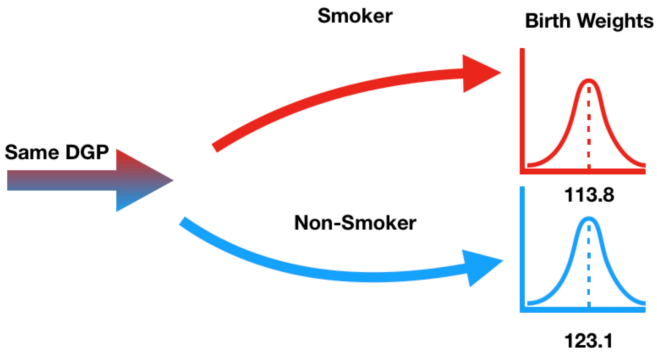
Loading [MathJax]/extensions/Safe.js



There appears to be a difference, but can it be attributed to random chance?

Null hypothesis: birth weights come from the *same* distribution

- Our null hypothesis states that "smoker" / "non-smoker" labels have no relationship to birth weight.
- In other words, the "smoker" / "non-smoker" labels **may well have** been assigned at random.

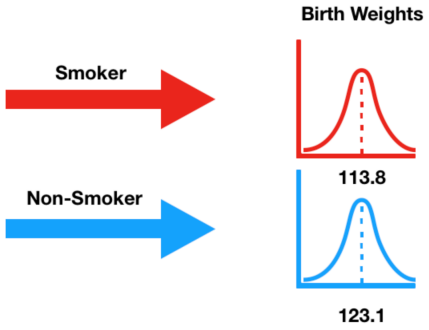


DGP stands for "data generating process" — think of this as another word for population.

Alternative hypothesis: birth weights come from *different* distributions

Loading [MathJax]/extensions/Safe.js

- Our alternative hypothesis states that the birth weights weights of smokers' babies and non-smokers' babies come from different population distributions.
  - That is, they come from different data generating processes.
- It also states that smokers' babies weigh significantly less.

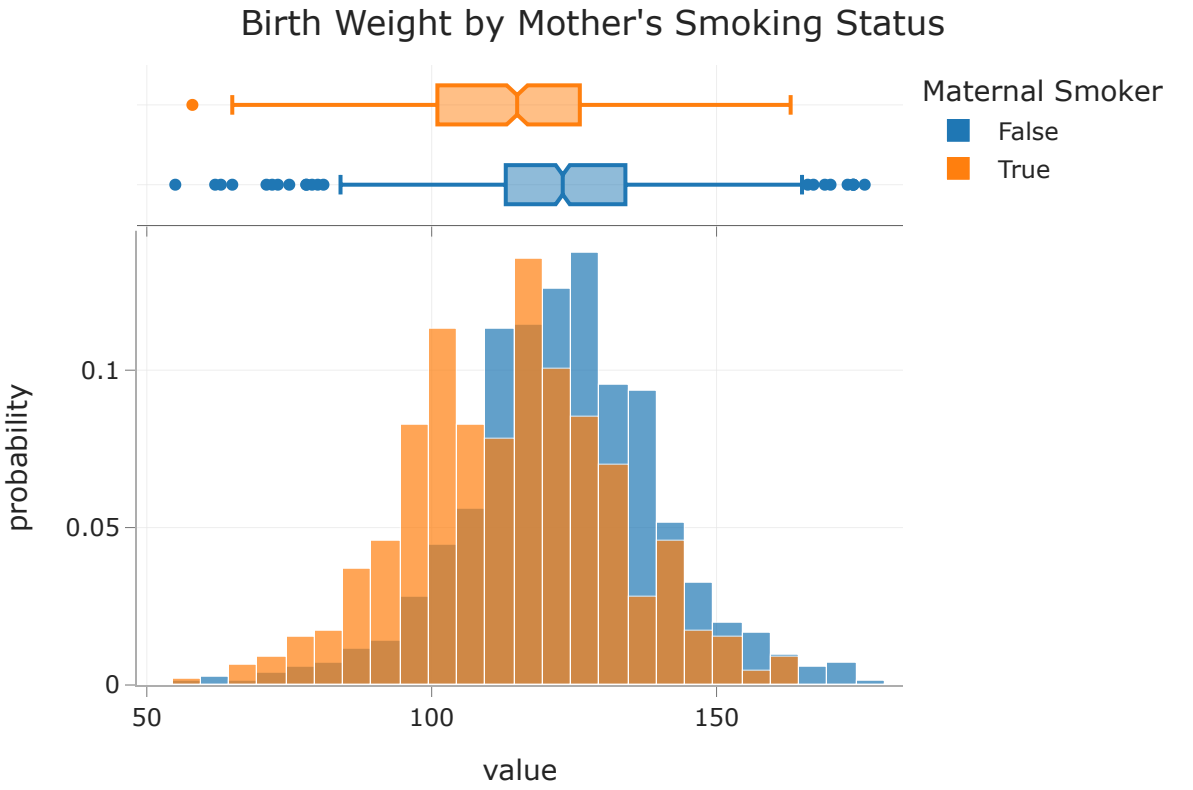


Choosing a test statistic

We need a test statistic that can measure **how different** two numerical distributions are.

In [66…

```
fig = px.histogram(baby, color='Maternal Smoker', histnorm='probability', r
                   title="Birth Weight by Mother's Smoking Status", barmode
fig
```



Easiest solution: Difference in group means.

Loading [MathJax]/extensions/Safe.js

Difference in group means

We'll choose our test statistic to be:

$$\text{mean weight of smokers' babies} - \text{mean weight of non-smokers' babies}$$

We could also compute the non-smokers' mean minus the smokers' mean, too.

In [67…
```
group_means = baby.groupby('Maternal Smoker')['Birth Weight'].mean()
group_means
```

Out[67]:
```
Maternal Smoker
False    123.09
True     113.82
Name: Birth Weight, dtype: float64
```

Use `loc` with `group_means` to compute the difference in group means.

In [68…
```
group_means.loc[True] - group_means.loc[False]
```

Out[68]:
```
np.float64(-9.266142572024918)
```

Hypothesis test setup

- **Null Hypothesis**: In the population, birth weights of smokers' babies and non-smokers' babies have the same distribution, and the observed differences in our samples are due to random chance.
- **Alternative Hypothesis**: In the population, smokers' babies have lower birth weights than non-smokers' babies, on average. The observed difference in our samples cannot be explained by random chance alone.
- **Test Statistic**: Difference in group means.

$$\text{mean weight of smokers' babies} - \text{mean weight of non-smokers' babies}$$

- **Issue**: We don't know what the population distribution actually is – so how do we draw samples from it?
  - This is different from the coin flipping, and the California ethnicity examples, because there **the null hypotheses were well-defined probability models**.

Implications of the null hypothesis

- Under the null hypothesis, both groups are sampled from the same distribution.
- If this is true, then the group label – `'Maternal Smoker'` – has no effect on the …ight.

Loading [MathJax]/extensions/Safe.js

- In our dataset, we saw **one assignment** of `True` or `False` to each baby.

In [69…
```
baby.head()
```

Out[69]:

	Maternal Smoker	Birth Weight
0	False	120
1	False	113
2	True	128
3	True	108
4	False	136

- Under the null hypothesis, we were just as likely to see **any other** assignment.

Permutation tests

- In a **permutation test**, we generate new data by **shuffling group labels**.
  - In our current example, this involves randomly assigning **babies to `True` or `False`**, while keeping the same number of `True`s and `False`s as we started with.
- On each shuffle, we'll compute our test statistic (difference in group means).
- If we shuffle many times and compute our test statistic each time, we will approximate the distribution of the test statistic.
- We can them compare our observed statistic to this distribution, as in any other hypothesis test.

Shuffling

- Our goal, by shuffling, is to randomly assign values in the `'Maternal Smoker'` column to values in the `'Birth Weight'` column.
- We can do this by shuffling either column **independently**.
- Easiest solution: `np.random.permutation`.
  - We could also use `df.sample`, but it's more complicated (and slower).

In [70…
```
np.random.permutation(baby['Birth Weight'])
```

Out[70]:
```
array([140, 113, 138, ..., 103, 142, 131])
```

Loading [MathJax]/extensions/Safe.js

In [71…
```
with_shuffled = baby.assign(Shuffled_Weights=np.random.permutation(baby['B
with_shuffled.head()
```

Out[71]:

	Maternal Smoker	Birth Weight	Shuffled_Weights
0	False	120	125
1	False	113	120
2	True	128	130
3	True	108	148
4	False	136	63

Now, we have a new sample of smokers' weights, and a new sample of non-smokers' weights!

Effectively, we took a random sample of 459 `'Birth Weights'` and assigned them to the smokers' group, and the remaining 715 to the non-smokers' group.

How close are the means of the shuffled groups?

One benefit of shuffling `'Birth Weight'` (instead of `'Maternal Smoker'`) is that grouping by `'Maternal Smoker'` allows us to see all of the following information with a single call to `groupby`.

In [72…
```
group_means = with_shuffled.groupby('Maternal Smoker').mean()
group_means
```

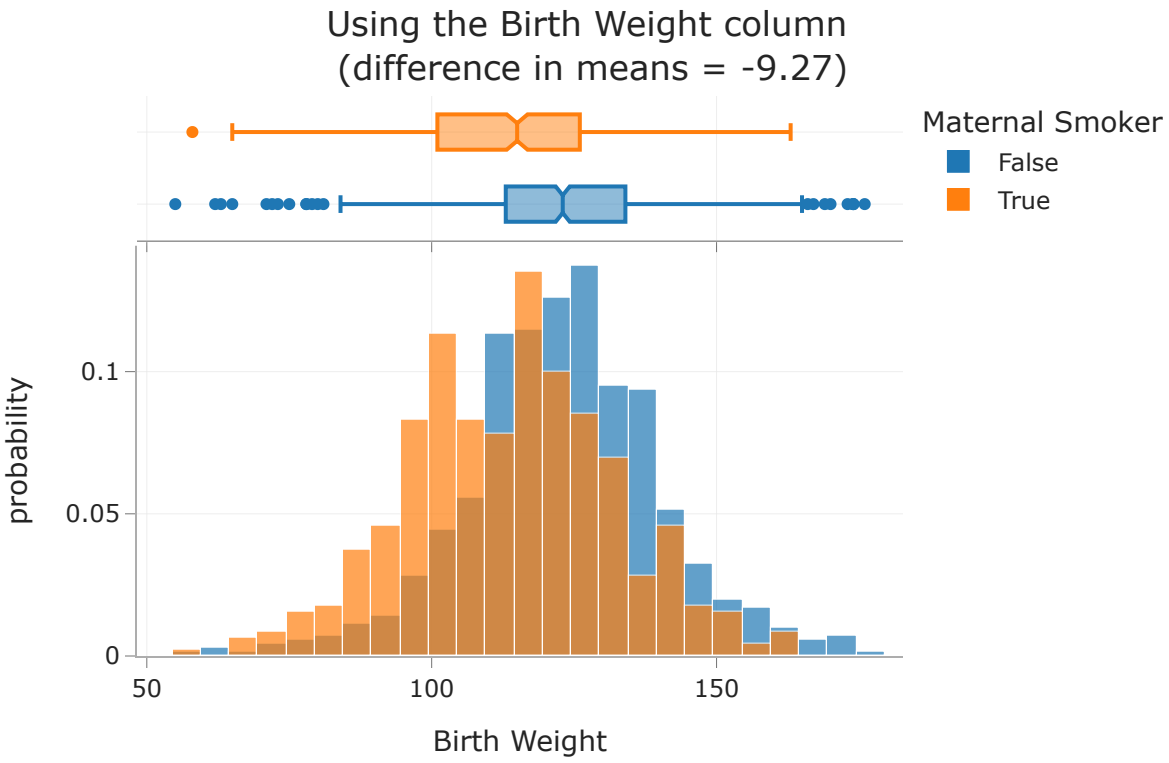
Out[72]:

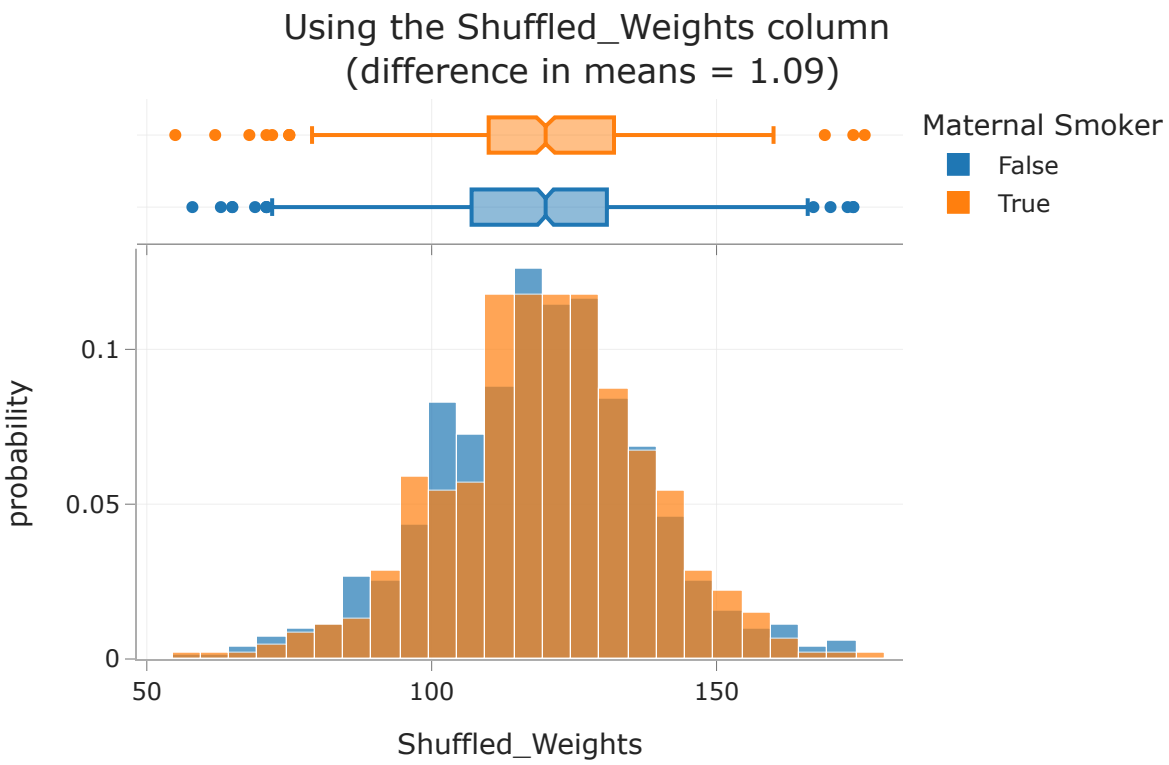
Maternal Smoker	Birth Weight	Shuffled_Weights
False	123.09	119.04
True	113.82	120.13

Let's visualize both pairs of distributions – what do you notice?

In [73…
```
for x in ['Birth Weight', 'Shuffled_Weights']:
    diff = group_means.loc[True, x] - group_means.loc[False, x]
    fig = px.histogram(
        with_shuffled, x=x, color='Maternal Smoker', histnorm='probability
        title=f"Using the {x} column <br>(difference in means = {diff:.2f}
        barmode='overlay', opacity=0.7)
    fig.update_layout(margin=dict(t=60))
    fig.show()
```

Loading [MathJax]/extensions/Safe.js





Simulating the empirical distribution of the test statistic

- This was just one random shuffle.
- The question we are trying to answer is, **how likely is it that a random shuffle results in two samples where the difference in group means (smoker minus**

Loading [MathJax]/extensions/Safe.js

non-smoker) is $\geq$ 9.26?

- To answer this question, we need the distribution of the test statistic. To generate that, we must shuffle many, many times. On each iteration, we must:

  1. Shuffle the weights and store them in a DataFrame.
  2. Compute the test statistic (difference in group means).
  3. Store the result.

In [74…
```
n_repetitions = 500

differences = []
for _ in range(n_repetitions):

    # Step 1: Shuffle the weights and store them in a DataFrame.
    with_shuffled = baby.assign(Shuffled_Weights=np.random.permutation(baby

    # Step 2: Compute the test statistic.
    # Remember, False (0) comes before True (1),
    # so this computes True - False.
    group_means = (
        with_shuffled
        .groupby('Maternal Smoker')
        .mean()
        .loc[:, 'Shuffled_Weights']
    )
    difference = group_means.loc[True] - group_means.loc[False]

    # Step 4: Store the result
    differences.append(difference)

differences[:10]
```

Out[74]:
```
[np.float64(0.1098313451254711),
 np.float64(3.103990127519552),
 np.float64(-1.6108262108262181),
 np.float64(1.630162865456981),
 np.float64(-1.753916236269177),
 np.float64(1.916342916342927),
 np.float64(0.07048158812864358),
 np.float64(0.7251184545302181),
 np.float64(-2.104486798604455),
 np.float64(-0.39456099456099025)]
```

We already computed the observed statistic earlier, but we compute it again below to keep all of our calculations together.

In [75…
```
mean_weights = baby.groupby('Maternal Smoker')['Birth Weight'].mean()
observed_difference = mean_weights[True] - mean_weights[False]
observed_difference
```

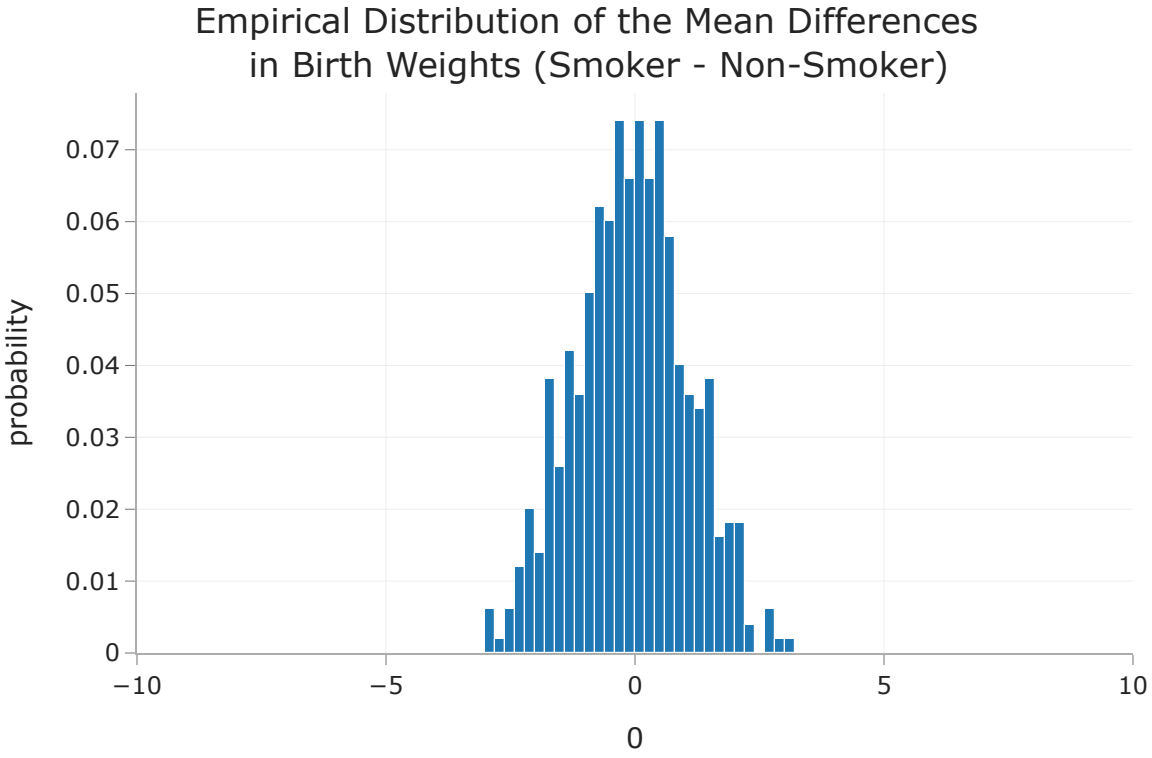
Out[75]:
```
np.float64(-9.266142572024918)
```

Loading [MathJax]/extensions/Safe.js

Conclusion of the test

In [76…

```
fig = px.histogram(
    pd.DataFrame(differences), x=0, nbins=50, histnorm='probability',
    title='Empirical Distribution of the Mean Differences <br> in Birth We
fig.add_vline(x=observed_difference, line_color='red')
fig.update_layout(xaxis_range=[-10, 10], margin=dict(t=60))
```



- Under the null hypothesis, we rarely see differences as large as 9.26 ounces.
- Therefore, **we reject the null hypothesis that the two groups come from the same distribution**. That is, the difference between the two samples is statistically significant.

⚠️ Caution!

- We **cannot** conclude that smoking **causes** lower birth weight!
- This was an observational study; there may be confounding factors.
  - Maybe smokers are more likely to drink caffeine, and caffeine causes lower birth weight.

Hypothesis testing vs. permutation testing

Loading [MathJax]/extensions/Safe.js ation tests **are** hypothesis tests!

- "Standard" hypothesis tests answer questions of the form:

> I have a population distribution, and I have one sample. Does this sample look like it was drawn from the population?

- Permutation tests answer questions of the form:

> I have two samples, but no information about any population distributions. Do these samples look like they were drawn from the same population? That is, do these two samples look "similar"?

Question 🤔 (Answer at dsc80.com/q)

Code: `gpa22`

This question was taken from the Spring 2022 Midterm Exam.

Consider the following pair of hypotheses.

- **Null Hypothesis**: The average GPA of UC San Diego admits from La Jolla Private is equal to the average GPA of UC San Diego admits from all schools.
- **Alternative Hypothesis**: The average GPA of UC San Diego admits from La Jolla Private is less than the average GPA of UC San Diego admits from all schools.

What type of test is this?

- A. Standard hypothesis test
- B. Permutation test

Question 🤔 (Answer at dsc80.com/q)

Code: `dist22`

This question was taken from the Spring 2022 Midterm Exam.

Consider the following pair of hypotheses.

- **Null Hypothesis**: The distribution of admitted, waitlisted, and rejected students at UC San Diego from Warren High is equal to the distribution of admitted, waitlisted, and rejected students at UC San Diego from La Jolla Private.
- **Alternative Hypothesis**: The distribution of admitted, waitlisted, and rejected students at UC San Diego from Warren High is different from the distribution of

Loading [MathJax]/extensions/Safe.jsd, waitlisted, and rejected students at UC San Diego from La Jolla Private.

What type of test is this?

- A. Standard hypothesis test
- B. Permutation test

Permutation testing meets TVD

> Note: This section has another hypothesis testing example. We might not have time to cover the example in lecture, but you should understand it.
>
> You can also watch **this podcast, starting from 4:43** for a walkthrough.

Example: Married vs. unmarried couples

- We will use data from a study conducted in 2010 by the National Center for Family and Marriage Research.
- The data consists of a national random sample of over 1,000 heterosexual couples who were either married or living together but unmarried.
- Each row corresponds to one **person** (not one couple).

In [77…
```
couples = pd.read_csv(Path('data') / 'married_couples.csv')
couples.head()
```

Out[77]:

	hh_id	gender	mar_status	rel_rating	...	education	hh_income	empl_status	h
0	0	1	1	1	...	12	14	1	
1	0	2	1	1	...	9	14	1	
2	1	1	1	1	...	11	15	1	
3	1	2	1	1	...	9	15	1	
4	2	1	1	1	...	12	14	1	

5 rows × 9 columns

In [78…
```
# What does this expression compute?
couples['hh_id'].value_counts().value_counts()
```

Out[78]:
```
count
2    1033
1       2
Name: count, dtype: int64
```

Loading [MathJax]/extensions/Safe.js

We won't use all of the columns in the DataFrame.

In [79…
```
couples = couples[['mar_status', 'empl_status', 'gender', 'age']]
couples.head()
```

Out[79]:

	mar_status	empl_status	gender	age
0	1	1	1	51
1	1	1	2	53
2	1	1	1	57
3	1	1	2	57
4	1	1	1	60

Cleaning the dataset

The numbers in the DataFrame correspond to the mappings below.

- `'mar_status'`: 1=married, 2=unmarried.
- `'empl_status'`: enumerated in the list below.
- `'gender'`: 1=male, 2=female.
- `'age'`: person's age in years.

In [80…
```
couples.head()
```

Out[80]:

	mar_status	empl_status	gender	age
0	1	1	1	51
1	1	1	2	53
2	1	1	1	57
3	1	1	2	57
4	1	1	1	60

In [81…
```
empl = [
    'Working as paid employee',
    'Working, self-employed',
    'Not working - on a temporary layoff from a job',
    'Not working - looking for work',
    'Not working - retired',
    'Not working - disabled',
    'Not working - other'
]
```

In [82…
```
couples = couples.replace({
    'mar_status': {1: 'married', 2: 'unmarried'},
    'gender': {1: 'M', 2: 'F'},
```

Loading [MathJax]/extensions/Safe.js

```
    'empl_status': {(k + 1): empl[k] for k in range(len(empl))}
})
```

In [83…
```
couples.head()
```

Out[83]:

	mar_status	empl_status	gender	age
0	married	Working as paid employee	M	51
1	married	Working as paid employee	F	53
2	married	Working as paid employee	M	57
3	married	Working as paid employee	F	57
4	married	Working as paid employee	M	60

Understanding the `couples` dataset

- Who is in our dataset? Mostly young people? Mostly married people? Mostly employed people?
- What is the distribution of values in each column?

In [84…
```
# For categorical columns, this shows the 10 most common values and their 
# For numerical columns, this shows the result of calling the .describe() 
for col in couples:
    if couples[col].dtype == 'object':
        empr = couples[col].value_counts(normalize=True).to_frame().iloc[:
    else:
        empr = couples[col].describe().to_frame()
    display(empr)
```

	proportion
mar_status	
married	0.72
unmarried	0.28

Loading [MathJax]/extensions/Safe.js

empl_status	proportion
Working as paid employee	0.61
Not working - other	0.10
Working, self-employed	0.10
Not working - looking for work	0.07
Not working - disabled	0.06
Not working - retired	0.05
Not working - on a temporary layoff from a job	0.02

gender	proportion
M	0.5
F	0.5

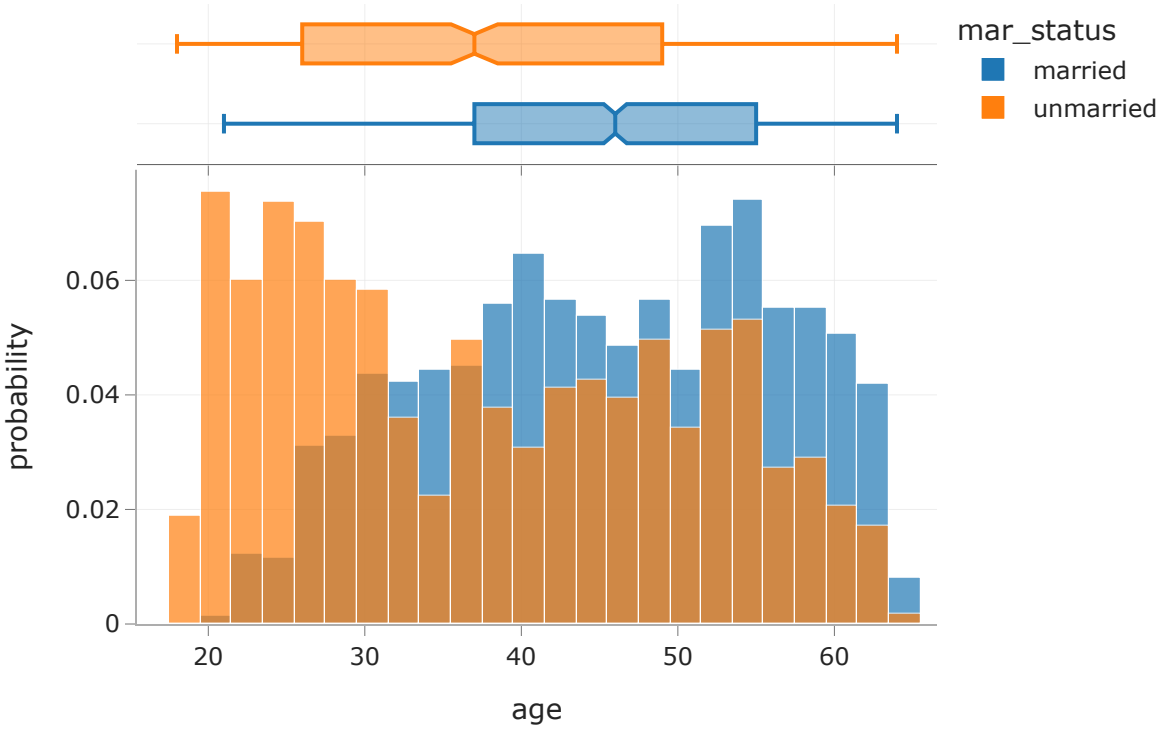
	age
count	2068.00
mean	43.17
std	11.91
...	...
50%	44.00
75%	53.00
max	64.00

8 rows × 1 columns

Let's look at the distribution of age **separately** for married couples and unmarried couples.

In [85…
```
px.histogram(couples, x='age', color='mar_status', histnorm='probability',
             barmode='overlay', opacity=0.7)
```

Loading [MathJax]/extensions/Safe.js



How are these two distributions different? Why do you think there is a difference?

Understanding employment status in households

- Do married households more often have a stay-at-home spouse?
- Do households with unmarried couples more often have someone looking for work?
- How much does the employment status of the different households vary?

To answer these questions, let's compute the distribution of employment status **conditional on household type (married vs. unmarried)**.

In [86…
```
couples.sample(5).head()
```

Out[86]:

	mar_status	empl_status	gender	age
1108	married	Working as paid employee	M	44
1970	unmarried	Working as paid employee	M	31
467	married	Working as paid employee	F	54
883	married	Not working - other	F	32
1082	married	Working as paid employee	M	47

In [87…
```
# Note that this is a shortcut to picking a column for values and using agg
= couples.pivot_table(index='empl_status', columns='mar_status',
```

Loading [MathJax]/extensions/Safe.js

```
empl_cnts
```

Out[87]:

mar_status	married	unmarried
empl_status		
Not working - disabled	72	45
Not working - looking for work	71	69
Not working - on a temporary layoff from a job	21	13
Not working - other	182	33
Not working - retired	94	11
Working as paid employee	906	347
Working, self-employed	138	66

Since there are a different number of married and unmarried couples in the dataset, we can't compare the numbers above directly. We need to convert counts to proportions, separately for married and unmarried couples.

In [88…

```
empl_cnts.sum()
```

Out[88]:

```
mar_status
married      1484
unmarried     584
dtype: int64
```

In [89…

```
cond_distr = empl_cnts / empl_cnts.sum()
cond_distr
```

Out[89]:

mar_status	married	unmarried
empl_status		
Not working - disabled	0.05	0.08
Not working - looking for work	0.05	0.12
Not working - on a temporary layoff from a job	0.01	0.02
Not working - other	0.12	0.06
Not working - retired	0.06	0.02
Working as paid employee	0.61	0.59
Working, self-employed	0.09	0.11

Both of the columns above sum to 1.

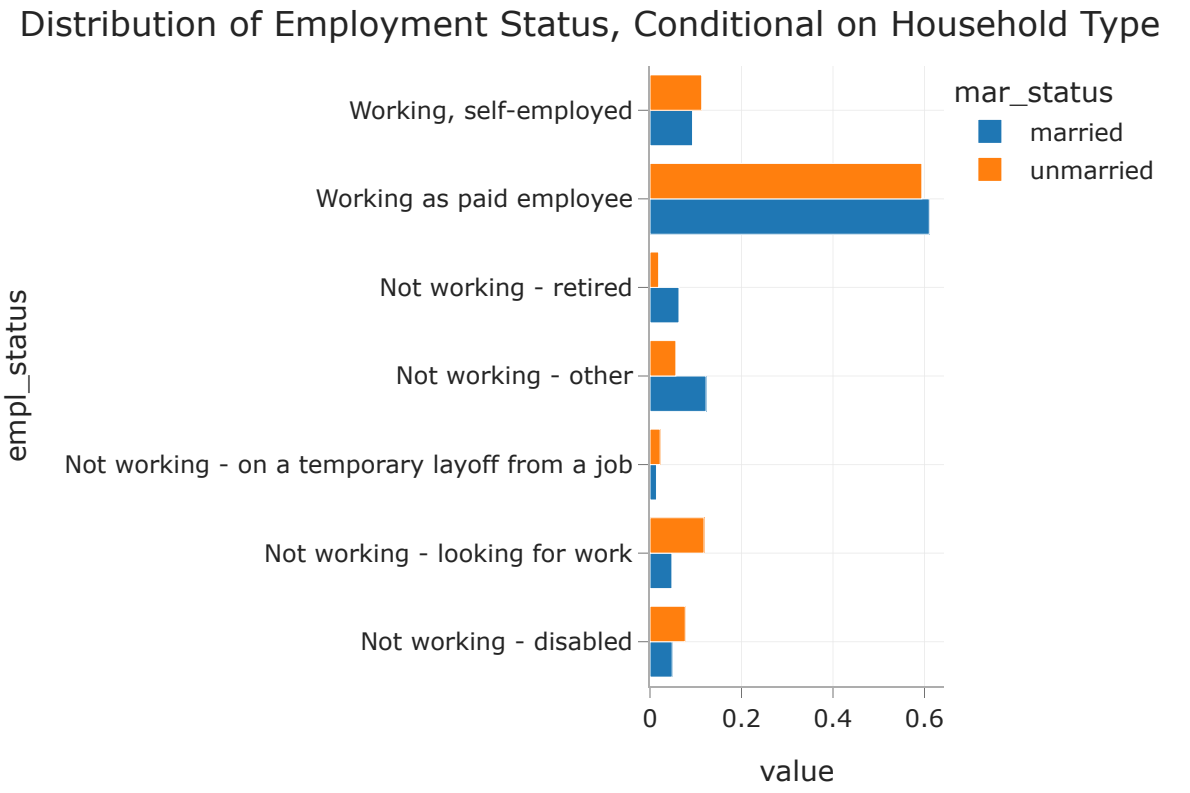
Differences in the distributions

Loading [MathJax]/extensions/Safe.js

Are the distributions of employment status for married people and for unmarried people who live with their partners **different**?

Is this difference just due to noise?

In [90…
```
cond_distr.plot(kind='barh', title='Distribution of Employment Status, Cond
```



Permutation test for household composition

- **Null Hypothesis**: In the US, the distribution of employment status among those who are married is the same as among those who are unmarried and live with their partners. The difference between the two observed samples is due to chance.
- **Alternative Hypothesis**: In the US, the distributions of employment status of the two groups are **different**.

Total variation distance

- Whenever we need to compare two categorical distributions, we use the TVD.
  - Recall, the TVD is the **sum of the absolute differences in proportions, divided by 2**.
- In DSC 10, the only test statistic we ever used in permutation tests was the difference in group means/medians, but the TVD can be used in permutation tests as well.

Loading [MathJax]/extensions/Safe.js

In [91…
```
cond_distr
```

Out[91]:

mar_status	married	unmarried
empl_status		
Not working - disabled	0.05	0.08
Not working - looking for work	0.05	0.12
Not working - on a temporary layoff from a job	0.01	0.02
Not working - other	0.12	0.06
Not working - retired	0.06	0.02
Working as paid employee	0.61	0.59
Working, self-employed	0.09	0.11

Let's first compute the observed TVD:

In [92…
```
(cond_distr['unmarried'] - cond_distr['married']).abs().sum() / 2
```

Out[92]:
```
np.float64(0.1269754089281099)
```

Since we'll need to calculate the TVD repeatedly, let's define a function that computes it.

In [93…
```
def tvd_of_groups(df, groups, cats):
    '''groups: the binary column (e.g. married vs. unmarried).
       cats: the categorical column (e.g. employment status).
    '''
    cnts = df.pivot_table(index=cats, columns=groups, aggfunc='size')
    # Normalize each column.
    distr = cnts / cnts.sum()
    # Compute and return the TVD.
    return (distr['unmarried'] - distr['married']).abs().sum() / 2
```

In [94…
```
# Same result as above.
observed_tvd = tvd_of_groups(couples, groups='mar_status', cats='empl_statu
observed_tvd
```

Out[94]:
```
np.float64(0.1269754089281099)
```

Simulation

- Under the null hypothesis, marital status is not related to employment status.
- We can shuffle the marital status column and get an equally-likely dataset.
- On each shuffle, we will compute the TVD.
- Once we have many TVDs, we can ask, **how often do we see a difference at s large as our observed difference?**

Loading [MathJax]/extensions/Safe.js

In [95…
```
couples.head()
```

Out[95]:

	mar_status	empl_status	gender	age
0	married	Working as paid employee	M	51
1	married	Working as paid employee	F	53
2	married	Working as paid employee	M	57
3	married	Working as paid employee	F	57
4	married	Working as paid employee	M	60

Here, we'll shuffle marital statuses, though remember, we could shuffle employment statuses too.

In [96…
```
couples.assign(shuffled_mar=np.random.permutation(couples['mar_status']))
```

Out[96]:

	mar_status	empl_status	gender	age	shuffled_mar
0	married	Working as paid employee	M	51	married
1	married	Working as paid employee	F	53	married
2	married	Working as paid employee	M	57	married
...	...	...	...	...	...
2065	unmarried	Working as paid employee	F	53	unmarried
2066	unmarried	Working as paid employee	M	44	unmarried
2067	unmarried	Working as paid employee	F	42	married

2068 rows × 5 columns

Let's do this repeatedly.

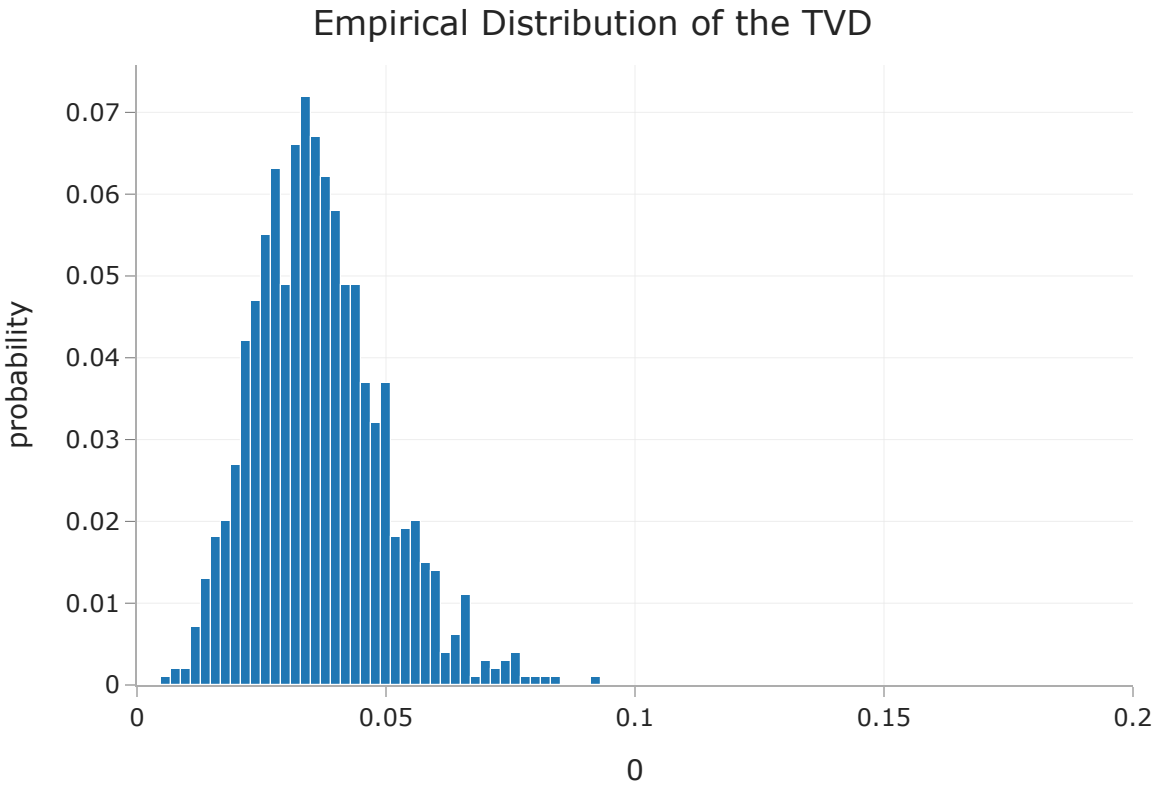
In [97…
```
N = 1000
tvds = []

for _ in range(N):
    # Shuffle marital statuses.
    with_shuffled = couples.assign(shuffled_mar=np.random.permutation(coup

    # Compute and store the TVD.
    tvd = tvd_of_groups(with_shuffled, groups='shuffled_mar', cats='empl_s
    tvds.append(tvd)
```

Notice that by defining a function that computes our test statistic, our simulation code is much cleaner.

Loading [MathJax]/extensions/Safe.js

Conclusion of the test

In [98…
```
fig = px.histogram(tvds, x=0, nbins=50, histnorm='probability',
                   title='Empirical Distribution of the TVD')
fig.update_layout(xaxis_range=[0, 0.2])
```



We **reject** the null hypothesis that married/unmarried households have similar employment makeups.

We can't say anything about **why** the employment makeups are different, though!

Summary, next time

Summary

- Both "standard" hypothesis tests and permutation tests are forms of hypothesis tests, which are used to assess whether some observation in our data looks **significant**.
  - See the end of the "Permutation testing" section for the distinction between the two.
- To run a hypothesis test, we need to choose a **test statistic** such that large observed values point to one hypothesis and small observed values point to the other. Examples we've seen:
  - Number of heads, absolute difference between number of heads and expected number of heads (coin-flipping example in the pre-lecture reading).
  - Total variation distance.

Loading [MathJax]/extensions/Safe.js

- Difference in group means.

Next time

Identifying different ways in which data can be missing. Don't worry, hypothesis testing will be revisited!

Loading [MathJax]/extensions/Safe.js