Data Quality and Processing

CMSE 830

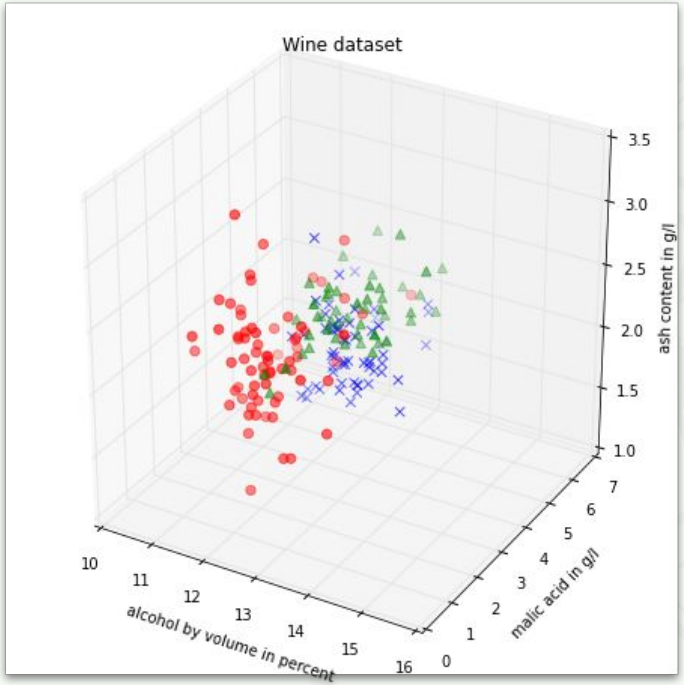
Attendance Sign In

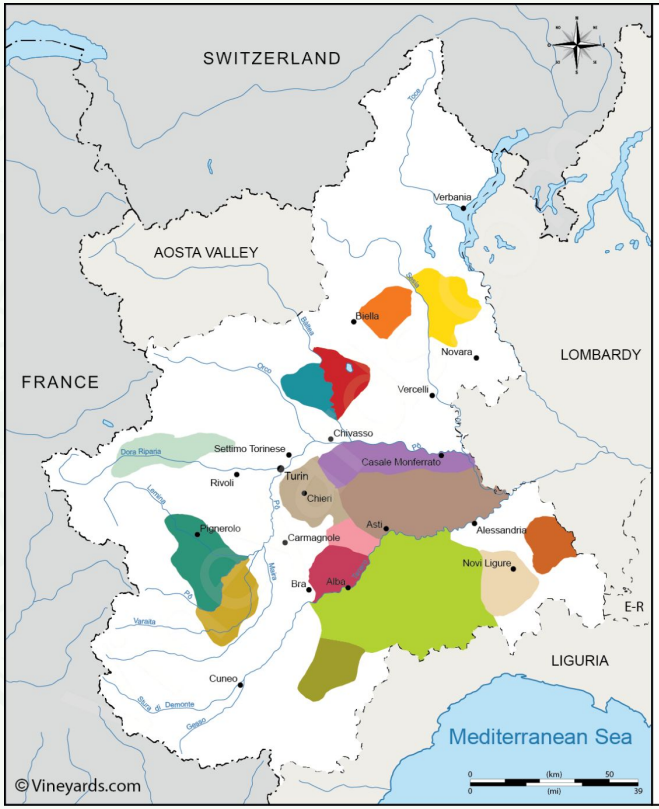
Prof. Murillo
Computational Mathematics, Science and Engineering
Michigan State University

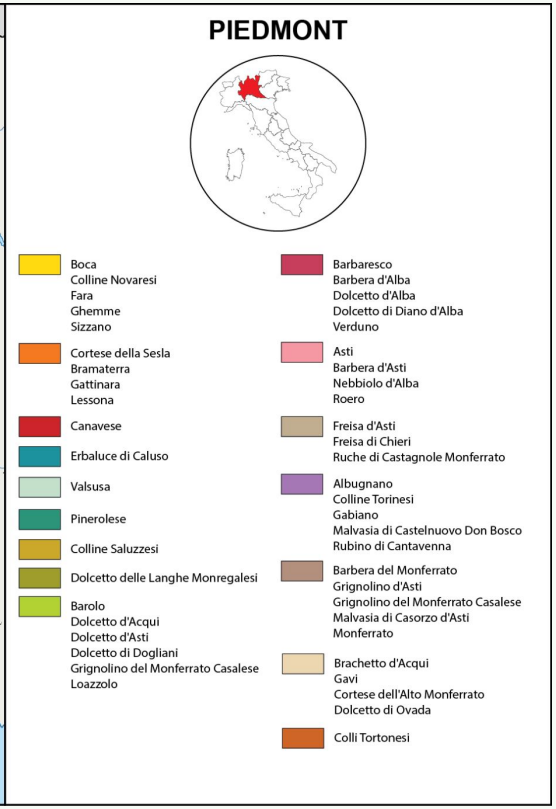
Plan For Today

1. discuss ICA
2. more on data types
3. processing data
4. Pandas?

ICA Discussion



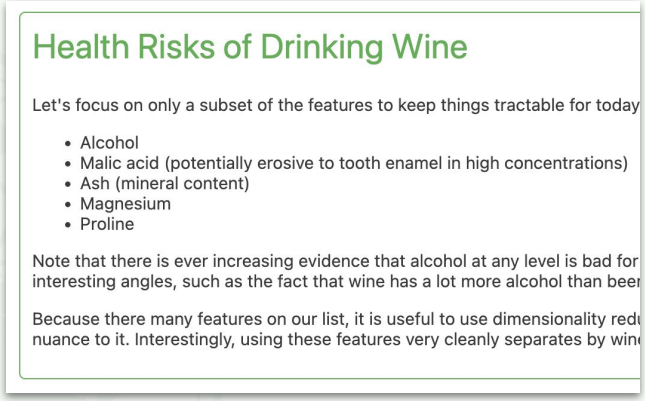




ICA Discussion

There were two goals of this ICA:

1. learn to *quickly* read the notebook *carefully* so that you can *pace* yourself - we went 10 minutes over
2. ***think*** about the data with no quantitative analysis



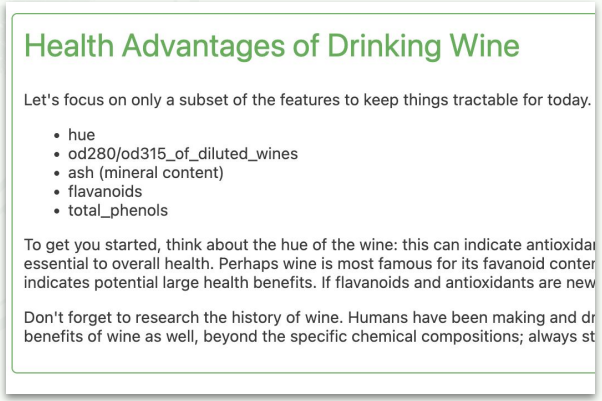
Health Risks of Drinking Wine

Let's focus on only a subset of the features to keep things tractable for today

- Alcohol
- Malic acid (potentially erosive to tooth enamel in high concentrations)
- Ash (mineral content)
- Magnesium
- Proline

Note that there is ever increasing evidence that alcohol at any level is bad for
interesting angles, such as the fact that wine has a lot more alcohol than bee

Because there many features on our list, it is useful to use dimensionality red
nuance to it. Interestingly, using these features very cleanly separates by win



Health Advantages of Drinking Wine

Let's focus on only a subset of the features to keep things tractable for today.

- hue
- od280/od315_of_diluted_wines
- ash (mineral content)
- flavanoids
- total_phenols

To get you started, think about the hue of the wine: this can indicate antioxida
essential to overall health. Perhaps wine is most famous for its favanoid conte
indicates potential large health benefits. If flavanoids and antioxidants are new

Don't forget to research the history of wine. Humans have been making and dr
benefits of wine as well, beyond the specific chemical compositions; always st

There are many numerical data types

Data Type	Description	Characteristics	Examples	Common Uses
Cardinal	Used for counting	- Whole numbers - Can be ordered - Can perform arithmetic	1, 2, 3, 10, 100	- Counting objects - Quantifying discrete items
Ordinal	Represents order or ranking	- Can be ordered - Differences between values may not be consistent	1st, 2nd, 3rd, 10th	- Ranking - Likert scales (e.g., 1-5 star ratings)
Interval	Measured along a scale with equal intervals	- Can be ordered - Equal intervals - No true zero point	Temperature in °C or °F	- Temperature measurements - Calendar years
Ratio	Has a true zero point and equal intervals	- Can be ordered - Equal intervals - Has a true zero point - Can calculate meaningful ratios	Height, Weight, Age	- Physical measurements - Financial data
Nominal (Numeric)	Numbers used as labels	- Cannot be ordered meaningfully - Cannot perform arithmetic	Zip codes, Phone numbers	- Identification - Categorization
Binary	Takes only two values	- Usually represented as 0 and 1 - Can represent yes/no or true/false	0/1, True/False	- Boolean logic - Digital systems
Continuous	Can take any value within a range	- Infinite possible values - Often measured with some precision	3.14159, 2.71828	- Scientific measurements - Physical properties
Discrete	Takes on distinct, separate values	- Finite or countably infinite values - Often whole numbers	Number of children, Dice rolls	- Counting occurrences - Probability

Each type needs to be handled differently because each contains information of a different character.

There are many numerical data types

Data Type	Description	Characteristics	Examples	Common Uses
Cardinal	Used for counting	- Whole numbers - Can be ordered - Can perform arithmetic	1, 2, 3, 10, 100	- Counting objects - Quantifying discrete items
Ordinal	Represents order or ranking	- Can be ordered - Differences between values may not be consistent	1st, 2nd, 3rd, 10th	- Ranking - Likert scales (e.g., 1-5 star ratings)
Interval	Measured along a scale with equal intervals	- Can be ordered - Equal intervals - No true zero point	Temperature in °C or °F	- Temperature measurements - Calendar years
Ratio	Has a true zero point and equal intervals	- Can be ordered - Equal intervals - Has a true zero point - Can calculate meaningful ratios	Height, Weight, Age	- Physical measurements - Financial data
Nominal (Numeric)	Numbers used as labels	- Cannot be ordered meaningfully - Cannot perform arithmetic	Zip codes, Phone numbers	- Identification - Categorization
Binary	Takes only two values	- Usually represented as 0 and 1 - Can represent yes/no or true/false	0/1, True/False	- Boolean logic - Digital systems
Continuous	Can take any value within a range	- Infinite possible values - Often measured with some precision	3.14159, 2.71828	- Scientific measurements - Physical properties
Discrete	Takes on distinct, separate values	- Finite or countably infinite values - Often whole numbers	Number of children, Dice rolls	- Counting occurrences - Probability

Each type needs to be handled differently because each contains information of a different character.

Ordinal data:

- corporal < sergeant < lieutenant < captain
- bachelor's < master's < PhD
- mild < medium < hot < very hot

Ordinal: Spiciness

Inconsistent Differences in Ordinal Data: Spiciness Levels

peppadew

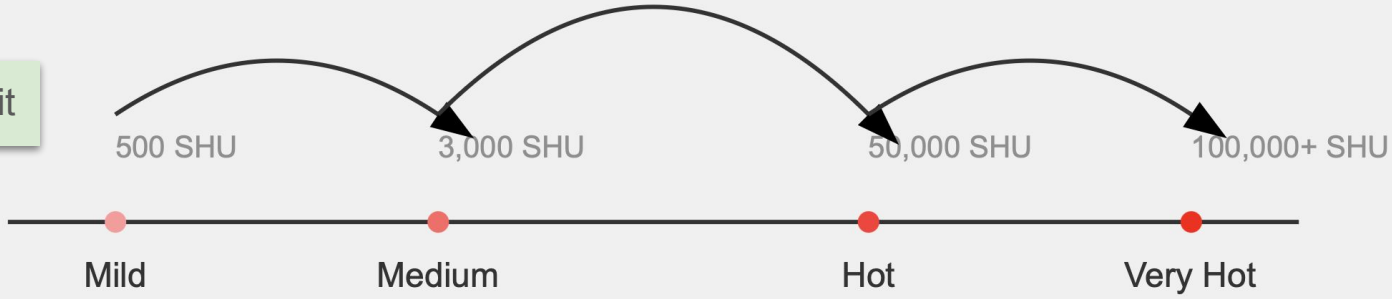
jalapeno

Thai Bird's Eye

habanero

Small Difference

Varies



SHU = Scoville Heat Unit

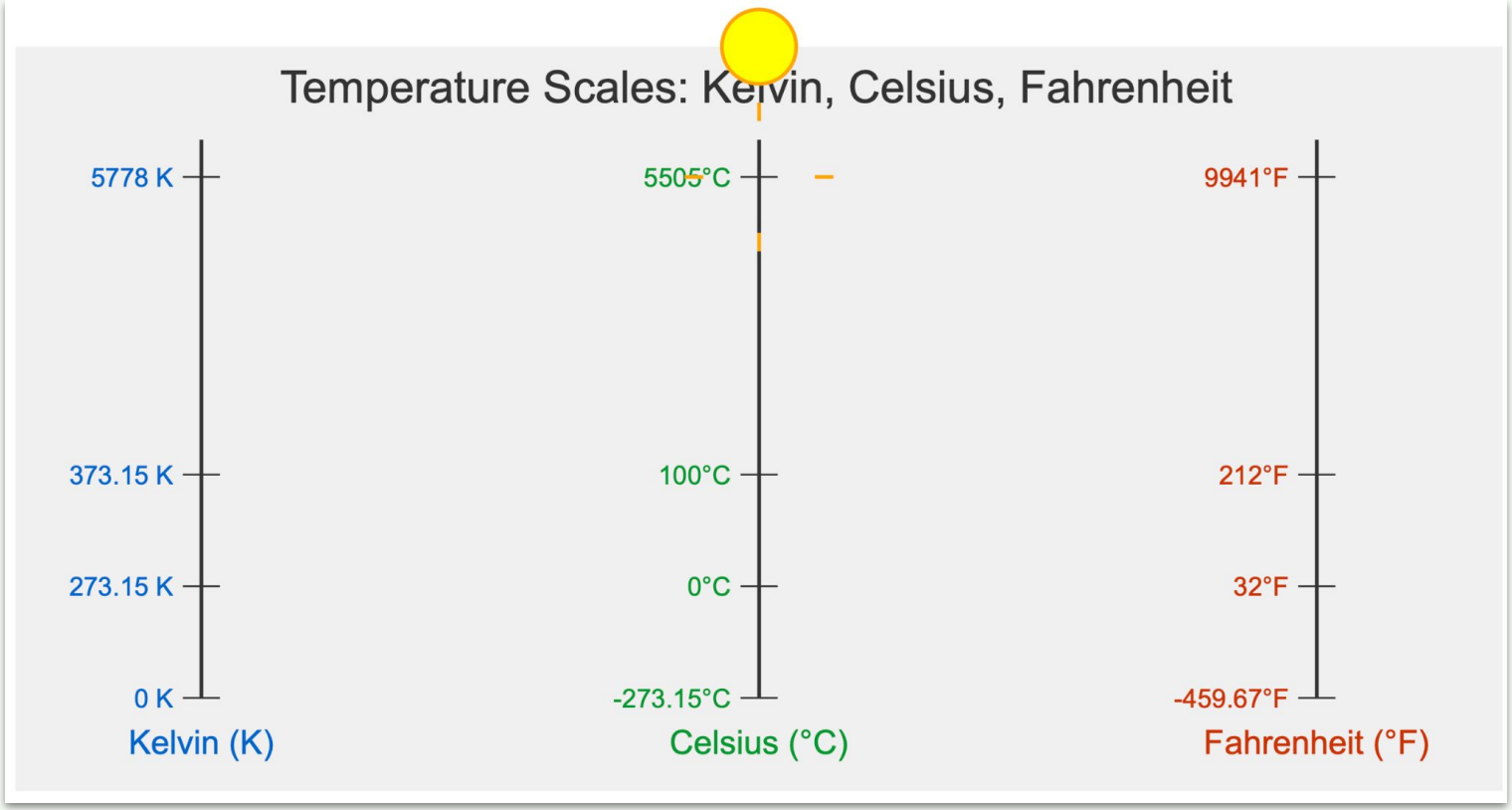
There are many numerical data types

Data Type	Description	Characteristics	Examples	Common Uses
Cardinal	Used for counting	- Whole numbers - Can be ordered - Can perform arithmetic	1, 2, 3, 10, 100	- Counting objects - Quantifying discrete items
Ordinal	Represents order or ranking	- Can be ordered - Differences between values may not be consistent	1st, 2nd, 3rd, 10th	- Ranking - Likert scales (e.g., 1-5 star ratings)
Interval	Measured along a scale with equal intervals	- Can be ordered - Equal intervals - No true zero point	Temperature in °C or °F	- Temperature measurements - Calendar years
Ratio	Has a true zero point and equal intervals	- Can be ordered - Equal intervals - Has a true zero point - Can calculate meaningful ratios	Height, Weight, Age	- Physical measurements - Financial data
Nominal (Numeric)	Numbers used as labels	- Cannot be ordered meaningfully - Cannot perform arithmetic	Zip codes, Phone numbers	- Identification - Categorization
Binary	Takes only two values	- Usually represented as 0 and 1 - Can represent yes/no or true/false	0/1, True/False	- Boolean logic - Digital systems
Continuous	Can take any value within a range	- Infinite possible values - Often measured with some precision	3.14159, 2.71828	- Scientific measurements - Physical properties
Discrete	Takes on distinct, separate values	- Finite or countably infinite values - Often whole numbers	Number of children, Dice rolls	- Counting occurrences - Probability

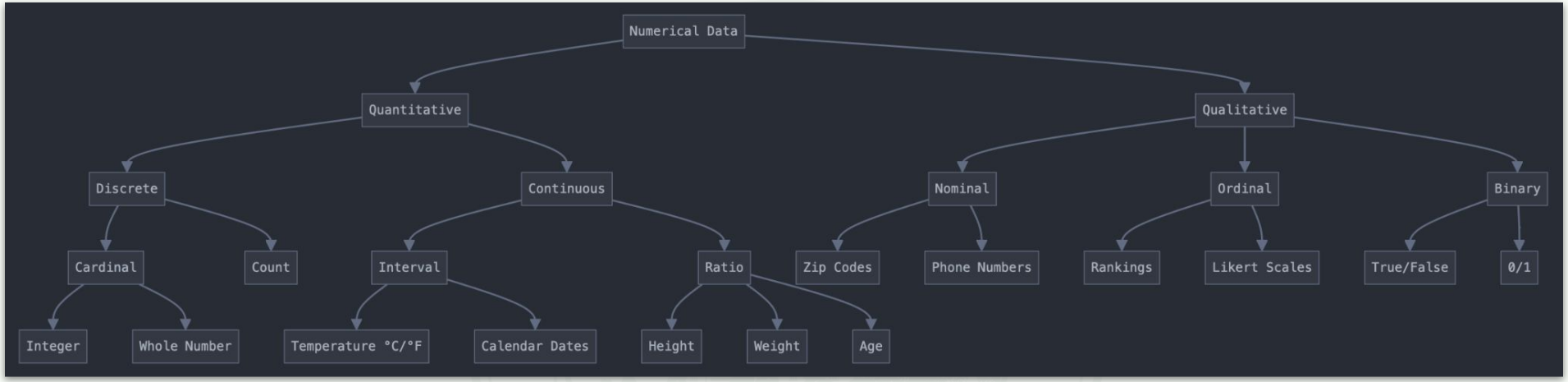
Each type needs to be handled differently because each contains information of a different character.

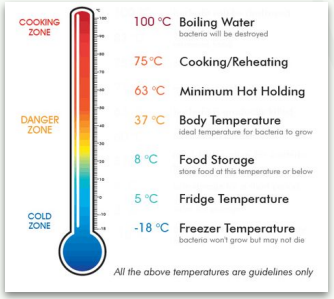
What is the temperature of the surface of the sun?

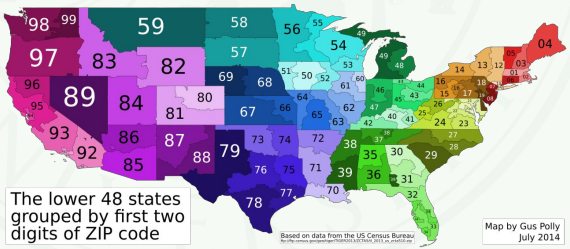
The answer very much depends on the units!

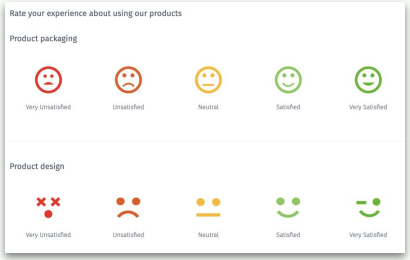


Organization of Numerical Data



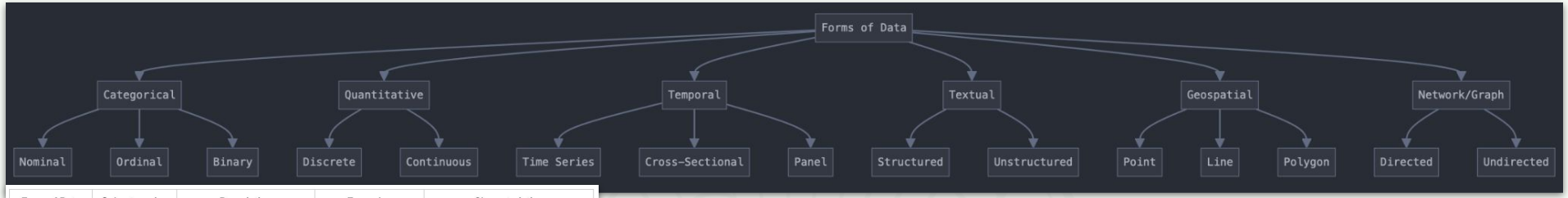




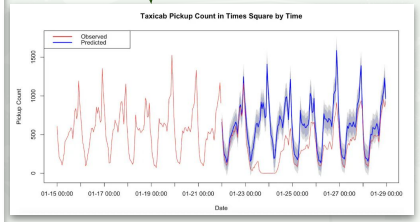


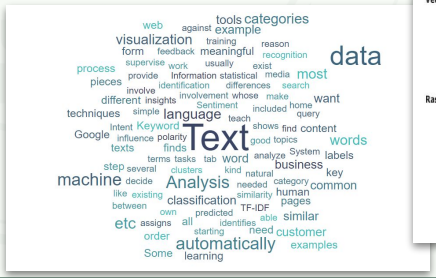
Attendance QR Code Slide... the correct one!

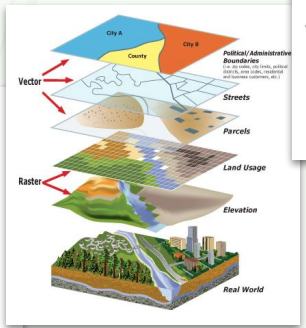
Forms of Data

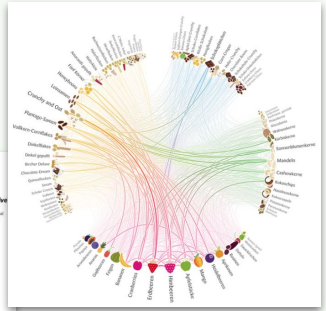


Form of Data	Subcategories	Description	Examples	Characteristics
Categorical	- Nominal - Ordinal - Binary	Data that can be divided into specific groups	- Colors - Ratings (1-5 stars) - Yes/No responses	- Cannot be used for numerical calculations - Can be counted and used for proportions
Quantitative	- Discrete - Continuous	Data that can be measured and has a numerical value	- Number of customers - Temperature - Height	- Can be used for mathematical operations - Can be discrete (countable) or continuous (measurable)
Temporal	- Time Series - Cross-Sectional - Panel	Data that represents points in time or durations	- Stock prices over time - Census data - GDP growth across countries over years	- Often used for trend analysis - Can be combined with other data types
Textual	- Structured - Unstructured	Data in the form of written or printed words	- Survey responses - Social media posts - Books	- Requires natural language processing techniques - Can be rich in information but challenging to analyze quantitatively
Geospatial	- Point - Line - Polygon	Data that has a geographic component	- GPS coordinates - Road networks - Country boundaries	- Often represented visually on maps - Requires specialized GIS tools for analysis
Network/Graph	- Directed - Undirected	Data that represents relationships between entities	- Social networks - Internet connections - Transportation routes	- Consists of nodes (entities) and edges (relationships) - Can reveal complex patterns and structures

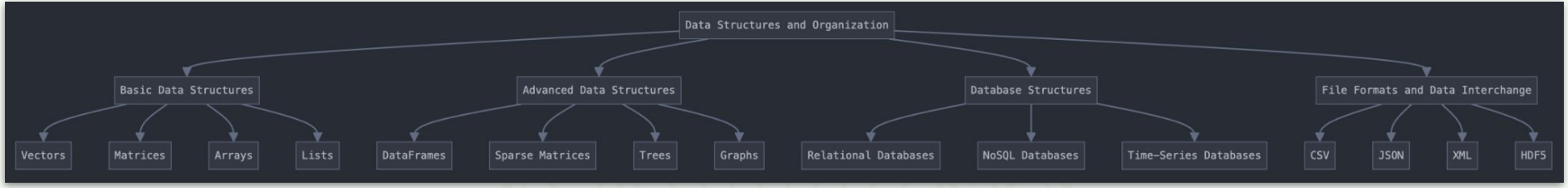








Data Structures: How Do We Store The Data?



Common Database Structures

Relational Database
(e.g., MySQL, PostgreSQL)

Users Table	Orders Table
ID | Name | Email	ID | UserID | Product

- Structured, tabular data
- Strong consistency
- Complex queries (SQL)

Document-based NoSQL
(e.g., MongoDB)

User Document
```
{
  "name": "John",
  "email": "j@ex.com",
}
```

Order Document
```
{
  "user": "John",
  "product": "Laptop"
}
```

- Flexible schema
- Scalability

Key-Value NoSQL
(e.g., Redis)

```
"user:1234" → {name: "John", ...}
"order:5678" → {product: "Laptop", ...}
"session:abcd" → {user: "1234", ...}
```

- Simple, fast lookups
- High performance

Time-Series Database
(e.g., InfluxDB)

```
timestamp | metric | value
2023-09-01 10:00 | temp | 22.5
2023-09-01 10:01 | temp | 22.7
2023-09-01 10:02 | temp | 22.6
```

- Optimized for time-based data
- Efficient for large volumes

Common Data File Formats

CSV
(Comma-Separated Values)

```
Name,Age,City
John,30,New York
Alice,25,London
```

- Simple, tabular data
- Easy to read and write

JSON
(JavaScript Object Notation)

```
{
  "name": "John",
  "age": 30,
  "city": "New York"
}
```

- Hierarchical data

XML
(eXtensible Markup Language)

```
<person>
  <name>John</name>
  <age>30</age>
  <city>New York</city>
</person>
```

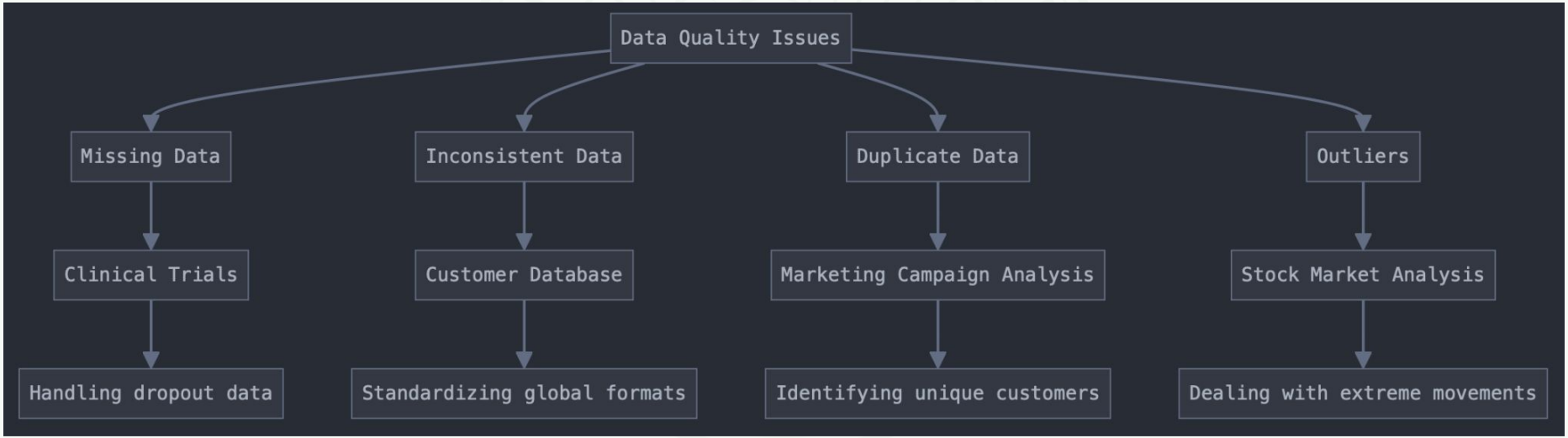
- Flexible, self-describing

HDF5
(Hierarchical Data Format)

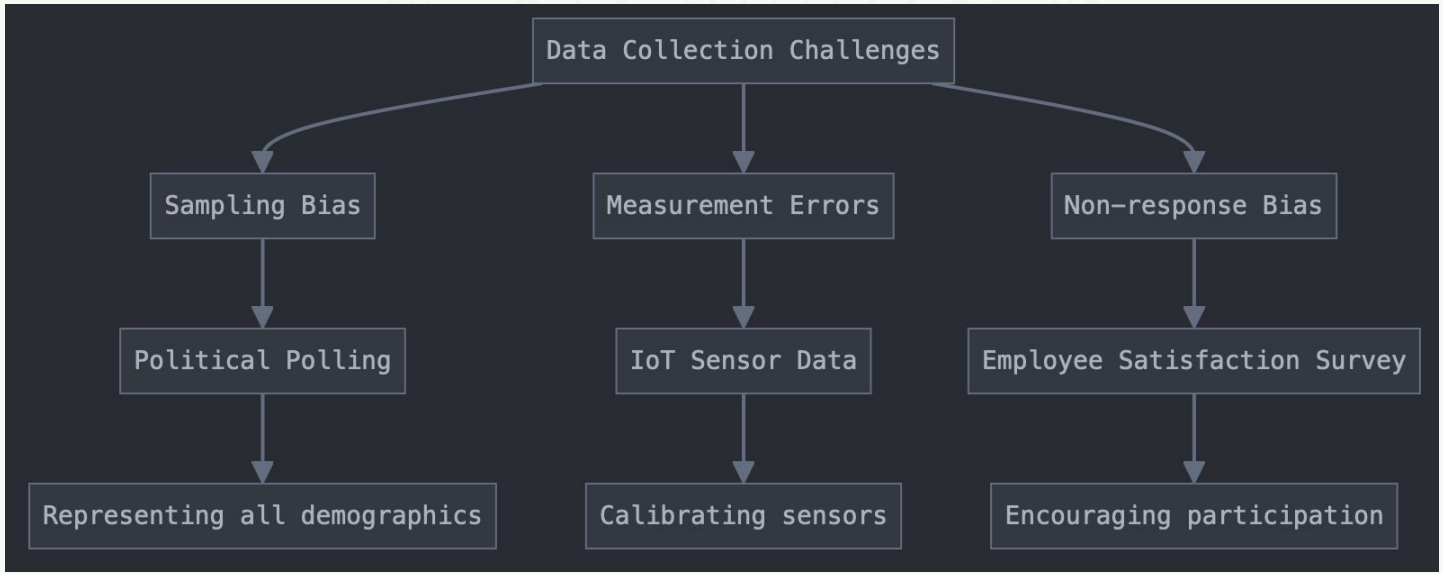
Hierarchical structure with groups and datasets (binary format)

- Efficient for large datasets
- Supports complex data types

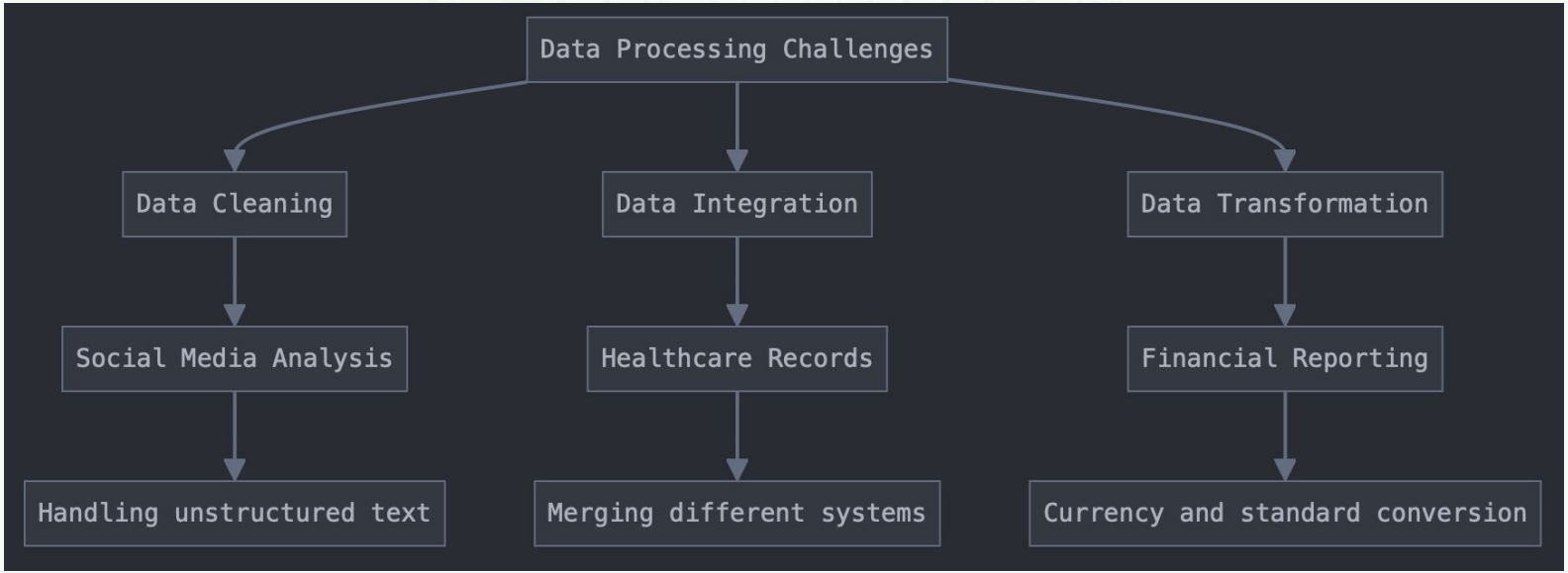
Issues With Data: Quality



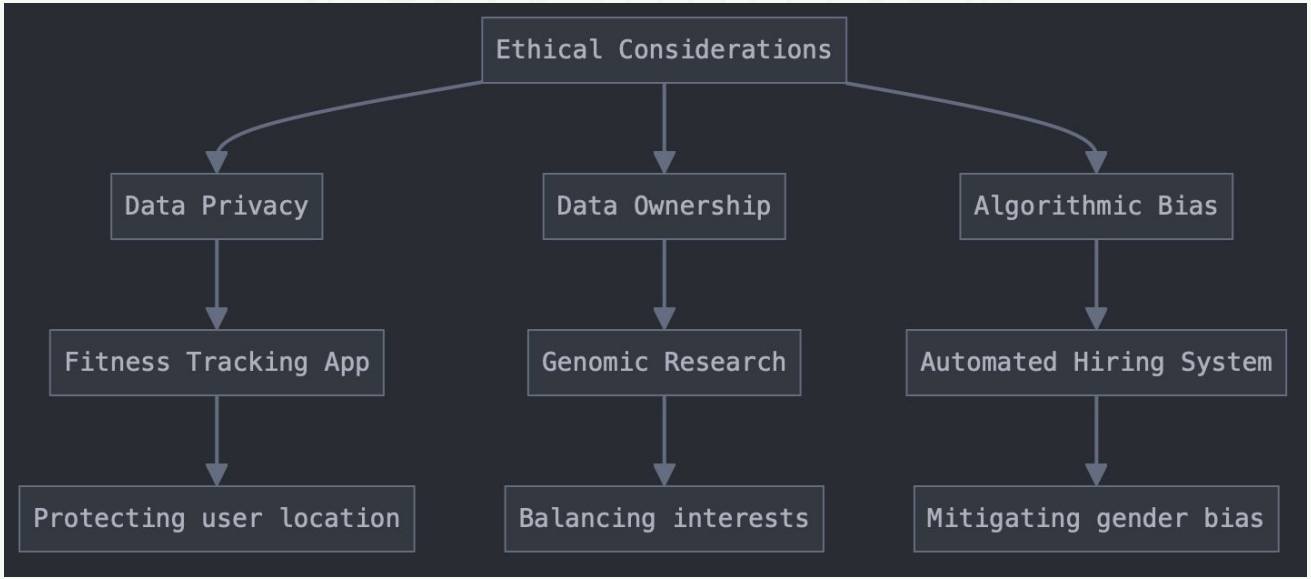
Issues With Data: Collection



Issues With Data: Processing



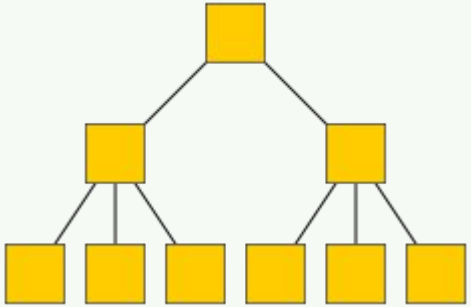
Issues With Data: Ethics



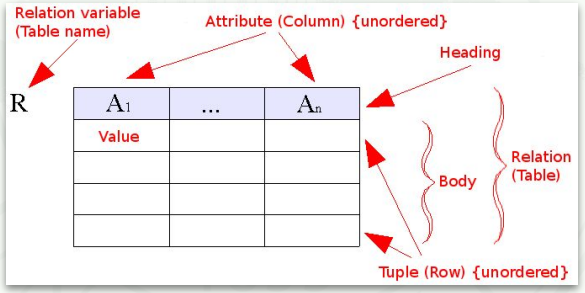
Data Model

A **data model** is an organization principle for the elements of a dataset.

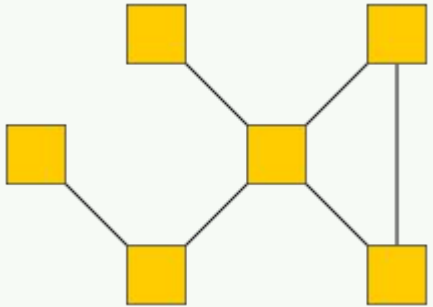
Some examples:



hierarchical



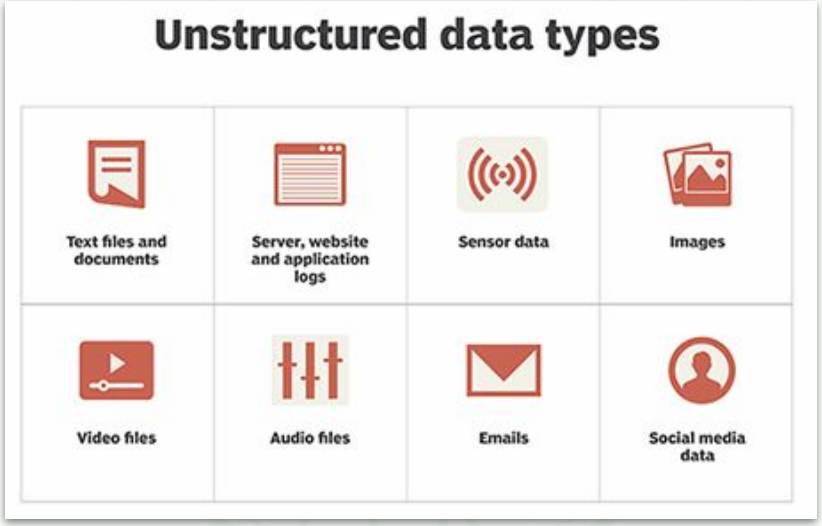
relational



network

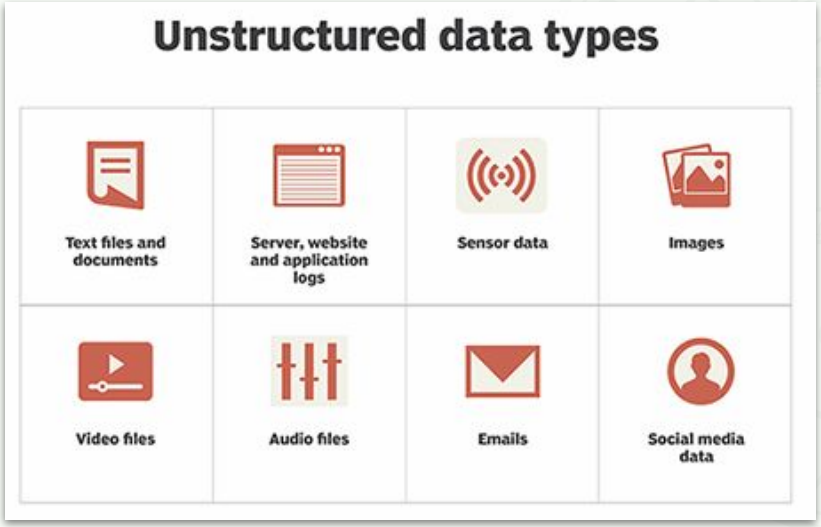
Structured and Unstructured Data

Unstructured data does not have a data model.



Structured and Unstructured Data

Unstructured data does not have a data model.



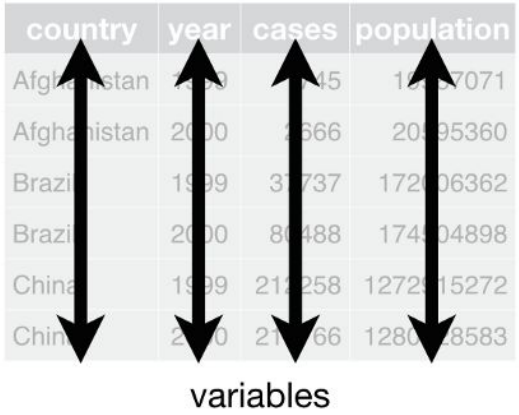
Structured data does have a data model.

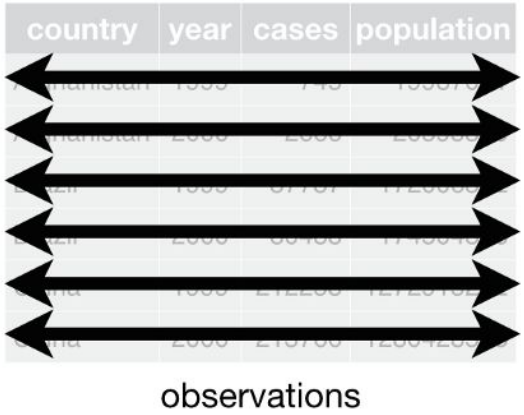
Structured data

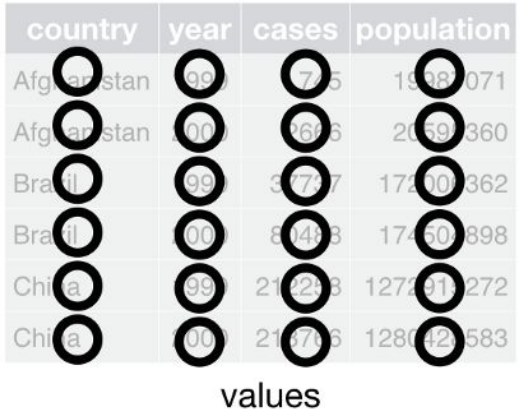
ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.

Tidy Data

Tidy data is defined as an organization or display of data such that columns are features and rows are samples.



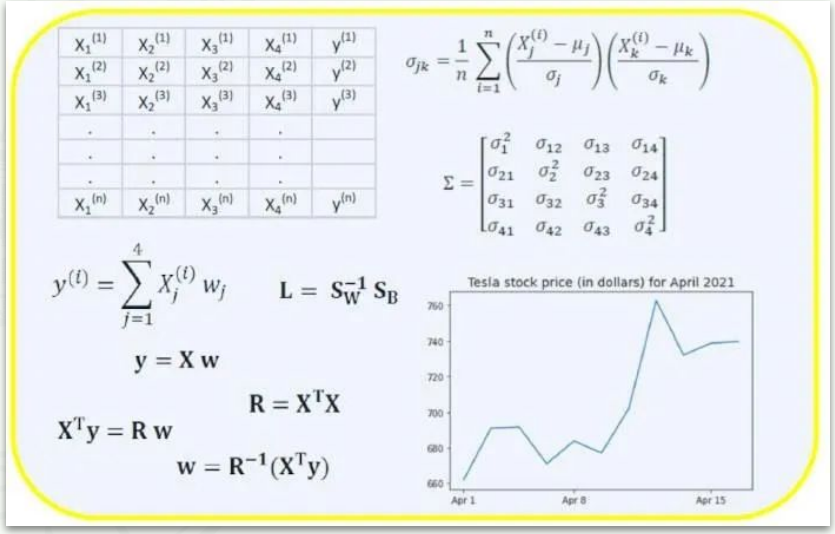




Data Matrix (Linear Algebra)

The **data matrix** is defined as a mathematical (i.e., linear algebra) matrix containing data organized in a tidy way.

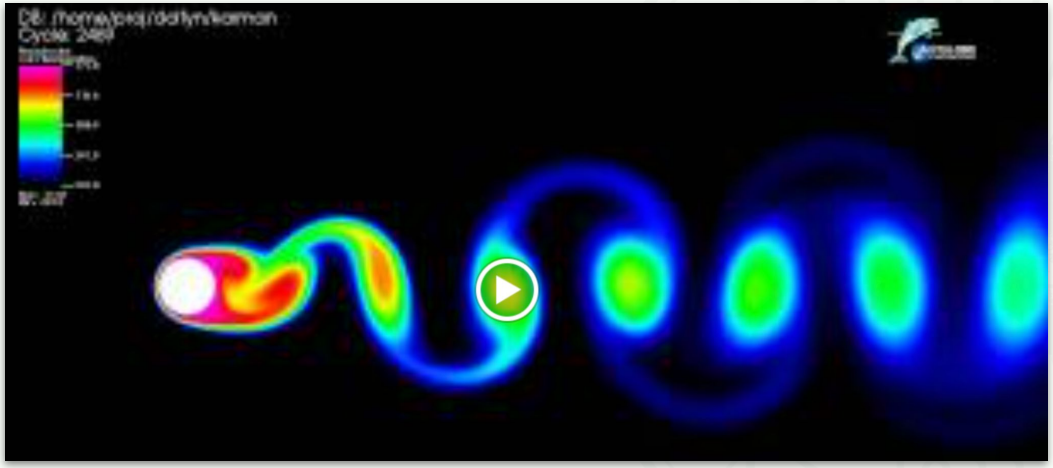
$X_1^{(1)}$	$X_2^{(1)}$	$X_3^{(1)}$	$X_4^{(1)}$
$X_1^{(2)}$	$X_2^{(2)}$	$X_3^{(2)}$	$X_4^{(2)}$
$X_1^{(3)}$	$X_2^{(3)}$	$X_3^{(3)}$	$X_4^{(3)}$
.	.	.	.
.	.	.	.
.	.	.	.
$X_1^{(n)}$	$X_2^{(n)}$	$X_3^{(n)}$	$X_4^{(n)}$



The data matrix will be covered in great detail later in the semester!

Spatiotemporal Data

Suppose we have data that varies in **space** and **time**.



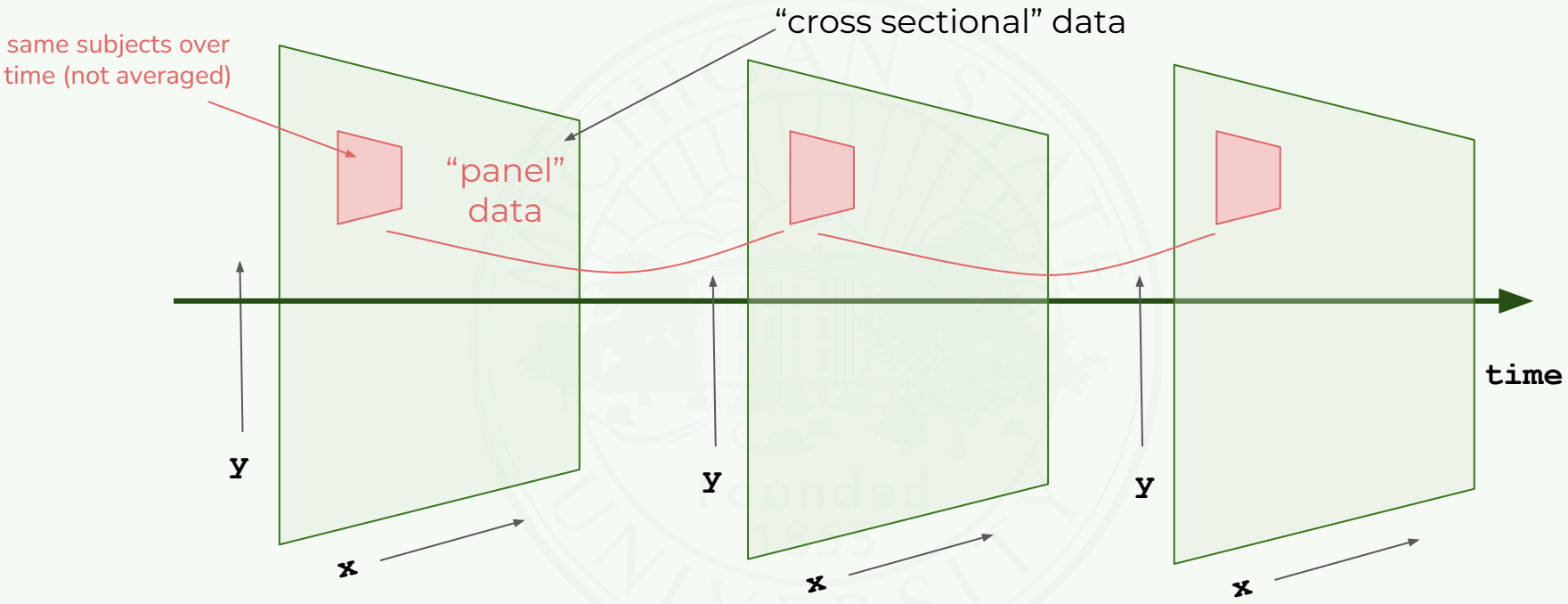
Here, x is the spatial data. Each column is a snapshot in time.

$$X = \begin{bmatrix} | & | & | \\ x(t_1) & x(t_2) & x(t_3) \\ | & | & | \end{bmatrix}$$

$$XX^T = \begin{bmatrix} | & | & | \\ x(t_1) & x(t_2) & x(t_3) \\ | & | & | \end{bmatrix} \begin{bmatrix} -x(t_1)- \\ -x(t_2)- \\ -x(t_3)- \end{bmatrix}$$

This yields the spatial covariance averaged over time.

Panel Data



"longitudinal" (time series) data

Detailed Processing of Data

Data preprocessing:

- scaling
  - normalizing
  - transforming
- encoding
- imputing

Scaling

Raw data typically contains numerical values that are in different ranges.

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	usa	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693	11.5	70	usa	buick skylark 320
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth satellite
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel sst
4	17.0	8	302.0	140.0	3449	10.5	70	usa	ford torino
5	15.0								ord galaxie 500
6	14.0								hevrolet impala
7	14.0								lymouth fury iii
8	14.0								ontiac catalina
9	15.0								mbassador dpl

```
1 df_m.describe()
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year
count	398.000000	398.000000	398.000000	392.000000	398.000000	398.000000	398.000000
mean	23.514573	5.454774	193.425879	104.469388	2970.424623	15.568090	76.010050
std	7.815984	1.701004	104.269838	38.491160	846.841774	2.757689	3.697627
min	9.000000	3.000000	68.000000	46.000000	1613.000000	8.000000	70.000000
25%	17.500000	4.000000	104.250000	75.000000	2223.750000	13.825000	73.000000
50%	23.000000	4.000000	148.500000	93.500000	2803.500000	15.500000	76.000000
75%	29.000000	8.000000	262.000000	126.000000	3608.000000	17.175000	79.000000
max	46.600000	8.000000	455.000000	230.000000	5140.000000	24.800000	82.000000

When modeling, the `weight` column may dominate the other features.

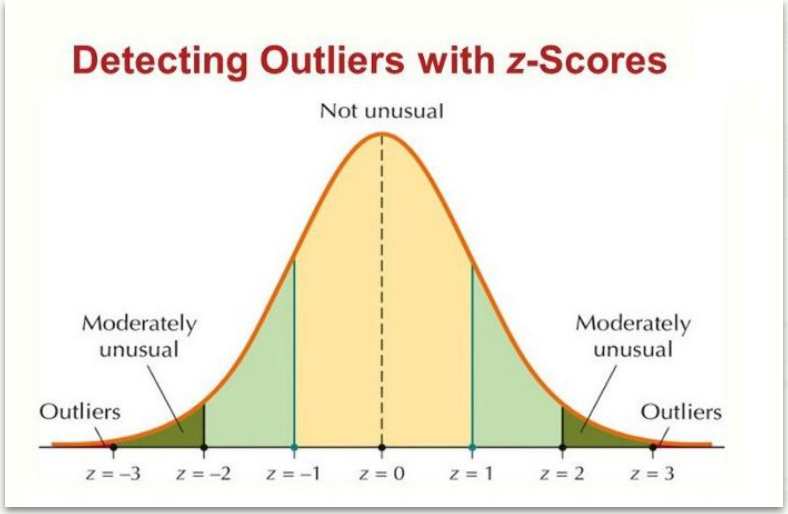
Z Scaling

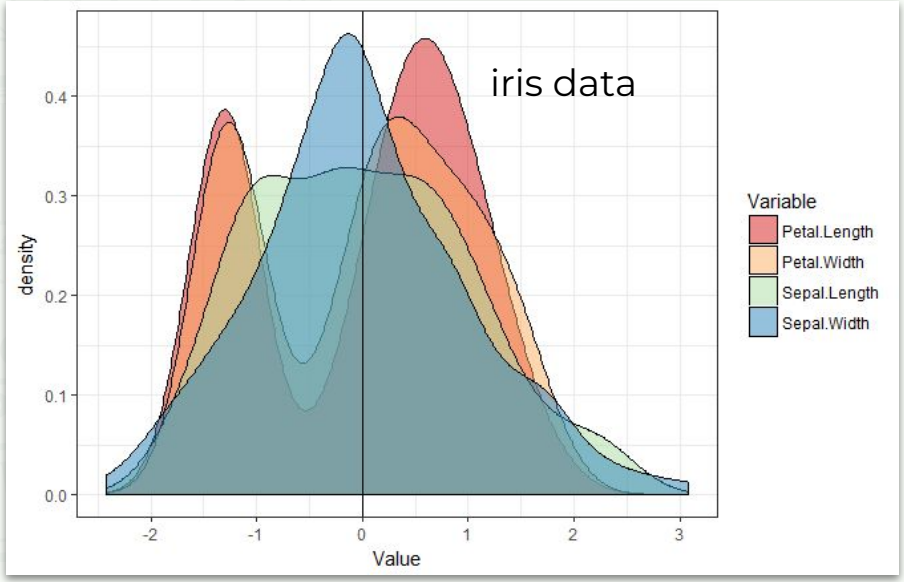
The standard way to scale data is through the *z* (standard) score:

$$z = \frac{x - \mu}{\sigma}$$

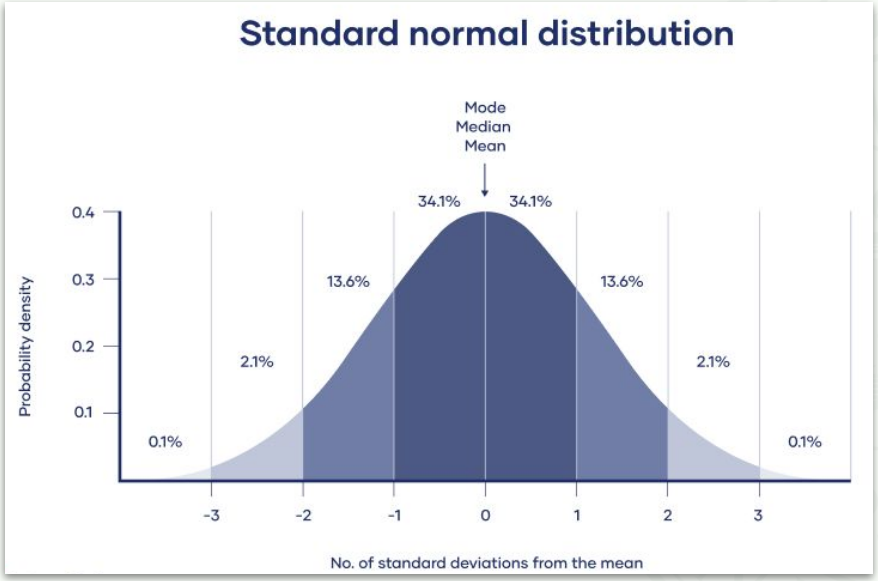
μ = mean

σ = standard deviation





Z Scaling: Connection to Gaussian/Normal



$$z = \frac{x - \mu}{\sigma}$$

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}$$

Although there is this interesting connection to the normal distribution, we apply the z-score to any distribution.

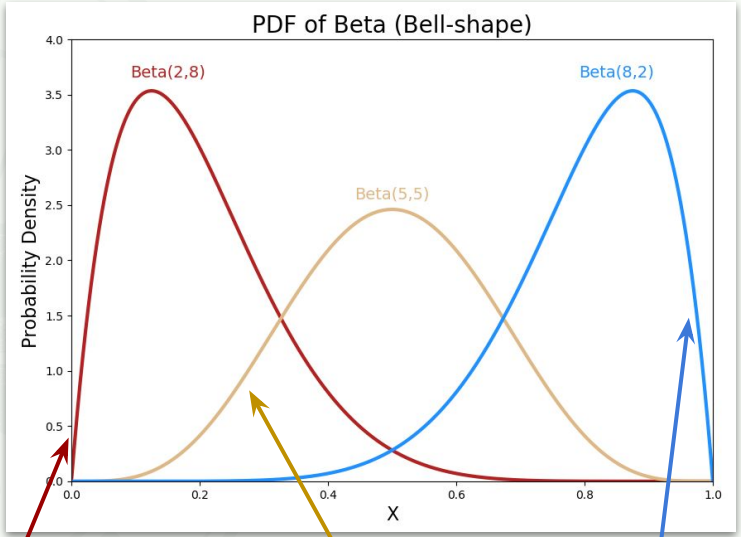
z-scaling Ranking Example

A committee is formed to read and rank submitted proposals.

Rankings are data used by a Program Officer to select proposals to be funded.

The z-score rescales these distributions so that they have comparable meanings.

Problem: people have personal approaches to ranking.

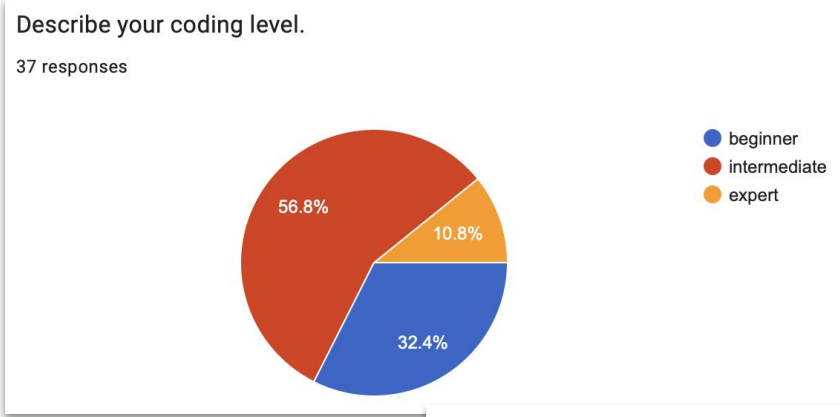


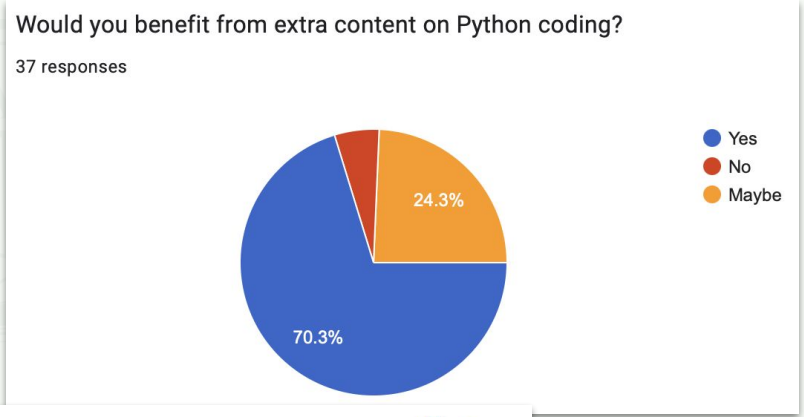
Tough person gives proposals low scores.

Balanced person gives a range of scores.

Nice person gives proposals high scores.

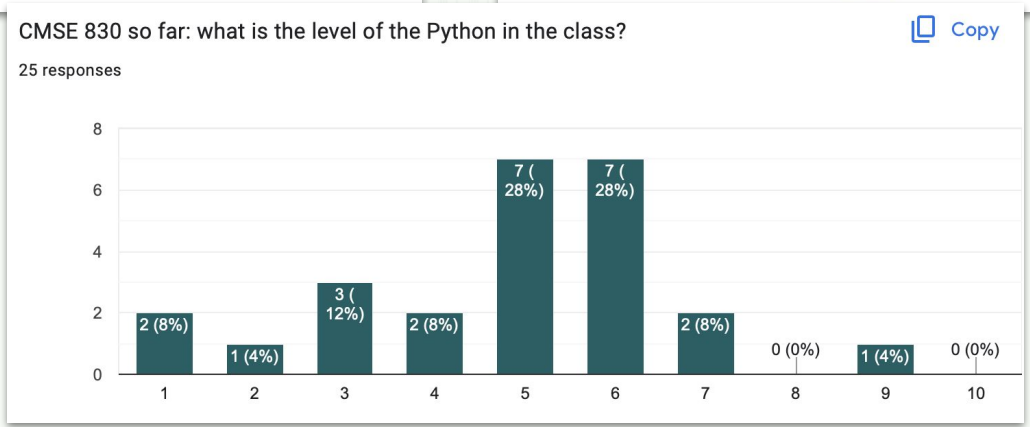
z-scaling Python Skill Example





The z-score could "re-normalize" the data.

But, I didn't collect names, so we can't do it with this dataset.



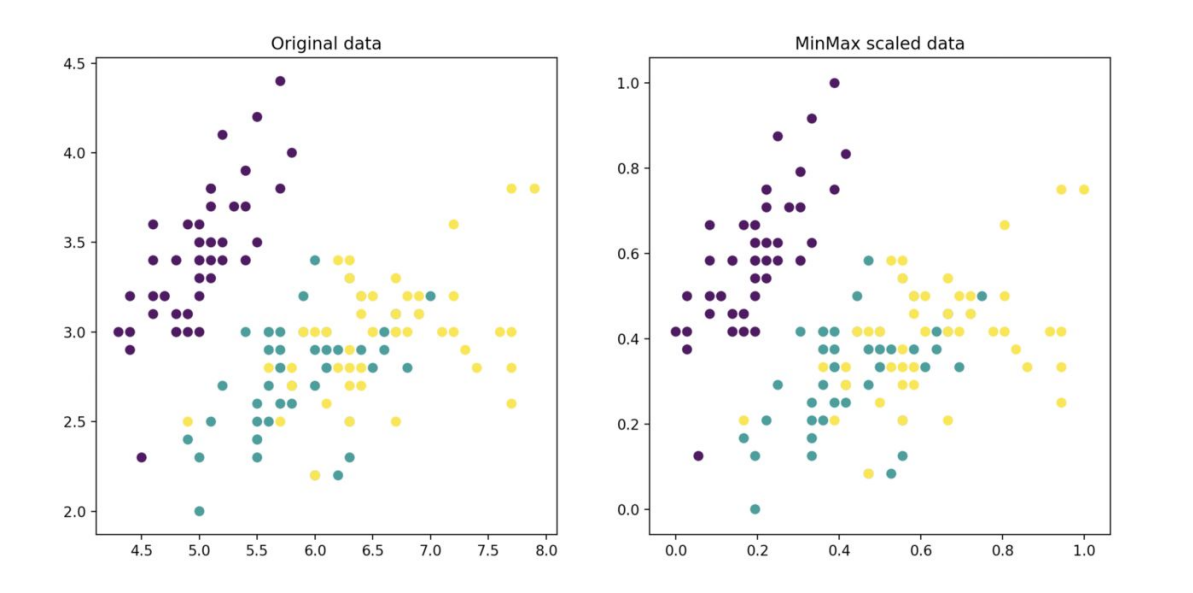
There are Many Scaling Methods

MinMax scaler maps the data into *[0,1]*.

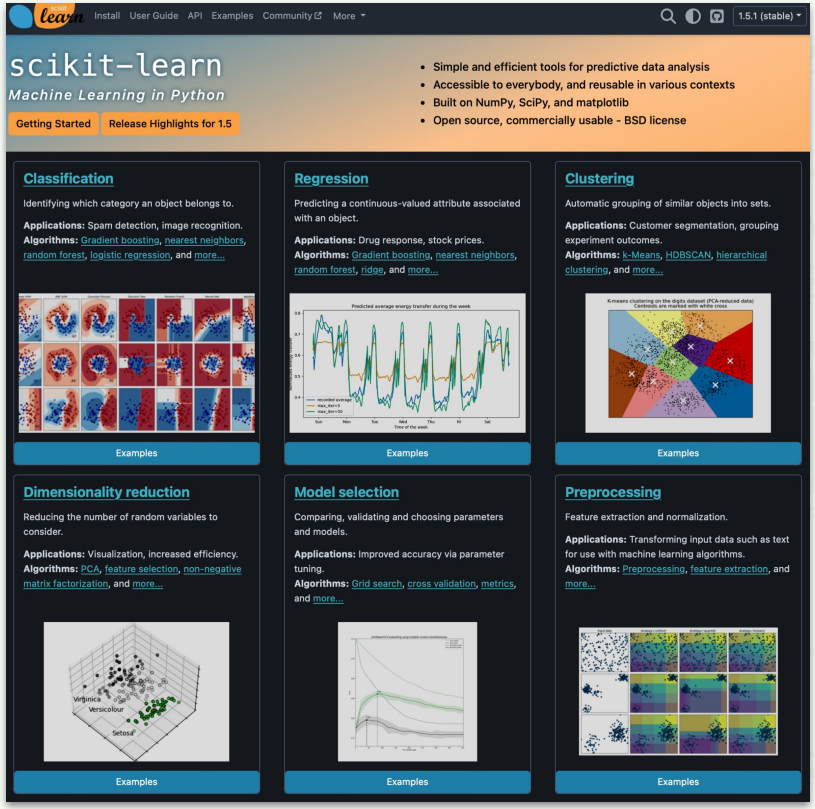
This can be advantageous when you can't handle negative values.

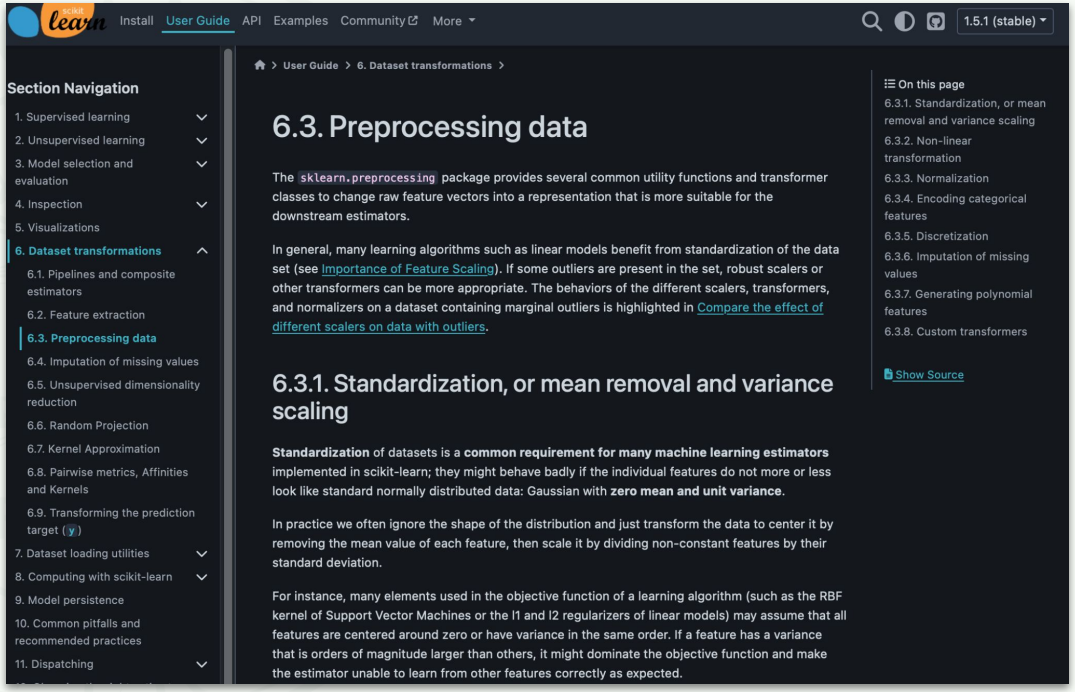
For example, you will take the logarithm of the data.

How you scale/normalize (or not) is problem dependent.



Interlude on `scikit-learn` and scaling





API of `StandardScaler`

import libraries

```
[6]  1  from sklearn.preprocessing import StandardScaler
     2  import numpy as np
     ✓  0.0s
```

fake data

```
[7]  1  X_train = np.array([[ 1., -1.,  2.],
     2                      [ 2.,  0.,  0.],
     3                      [ 0.,  1., -1.]])
     ✓  0.0s
```

check

```
[8]  1  X_train
     ✓  0.0s
...  array([[ 1., -1.,  2.],
            [ 2.,  0.,  0.],
            [ 0.,  1., -1.]])
```

create object

fit object to data

```
[18] 1  scaler = StandardScaler()
     2  scaler.fit(X_train)
     ✓  0.0s
```

μ

```
[12] 1  scaler.mean_
     ✓  0.0s
...  array([0.81649658, 0.81649658, 1.24721913])
```

σ

```
[13] 1  scaler.scale_
     ✓  0.0s
...  array([0.81649658, 0.81649658, 1.24721913])
```

apply to data and create new array

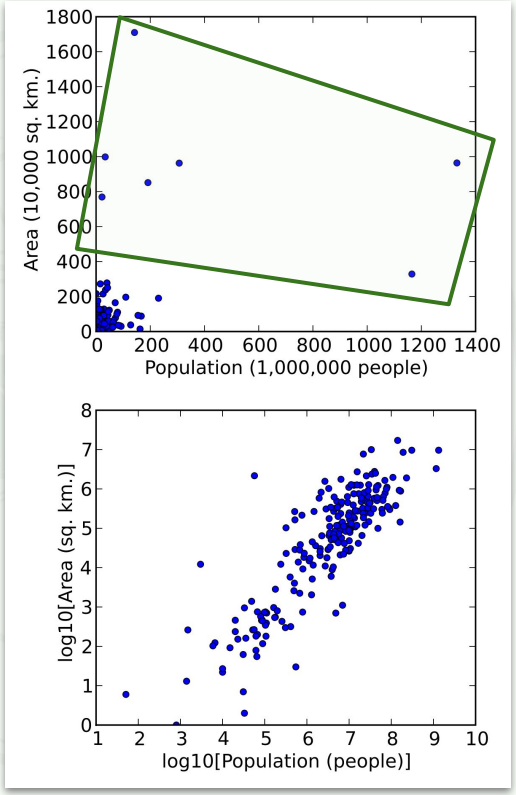
```
[16] 1  X_scaled = scaler.transform(X_train)
     ✓  0.0s
```

check

```
[17] 1  X_scaled
     ✓  0.0s
...  array([[ 0.        , -1.22474487,  1.33630621],
            [ 1.22474487,  0.        , -0.26726124],
            [-1.22474487,  1.22474487, -1.06904497]])
```

Transforming

Transforming is a related operation, and might be used with scaling.

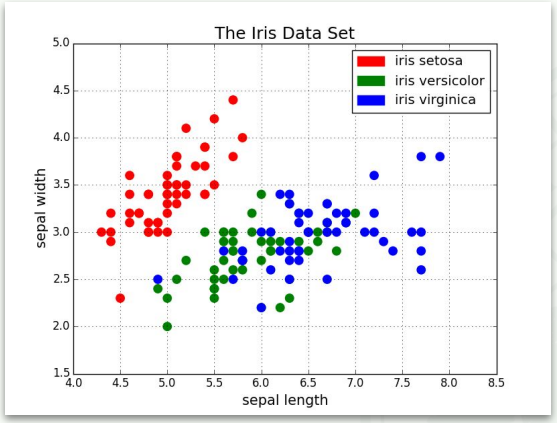


models will be dominated by these values

z-scaling doesn't help here

log-log transformed: much more balanced

Encoding



How do we handle categorical data?

Encoding maps the category onto a numerical value.

There are many ways to do this - very problem dependent.

- **label encoder**: encode target labels with *[0,N-1]* *(lossy)*
- **ordinal encoder**: features converted to ordinal numbers *[0,N-1]*
- **one-hot encoder**: creates binary feature for each category

Compare Label and One-Hot Encoding

Label Encoding

Food Name	Categorical #	Calories
Apple	1	95
Chicken	2	231
Broccoli	3	50

One Hot Encoding

Apple	Chicken	Broccoli	Calories
1	0	0	95
0	1	0	231
0	0	1	50

Missing Data: What To Do???

Let's examine the mpg dataset:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	usa	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693	11.5	70	usa	buick skylark 320
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth satellite
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel sst
4	17.0	8	302.0	140.0	3449	10.5	70	usa	ford torino
...	...	...	...	...	...	...	...	...	...
393	27.0	4	140.0	86.0	2790	15.6	82	usa	ford mustang gl
394	44.0	4	97.0	52.0	2130	24.6	82	europe	vw pickup
395	32.0	4	135.0	84.0	2295	11.6	82	usa	dodge rampage
396	28.0	4	120.0	79.0	2625	18.6	82	usa	ford ranger
397	31.0	4	119.0	82.0	2720	19.4	82	usa	chevy s-10

398 rows × 9 columns

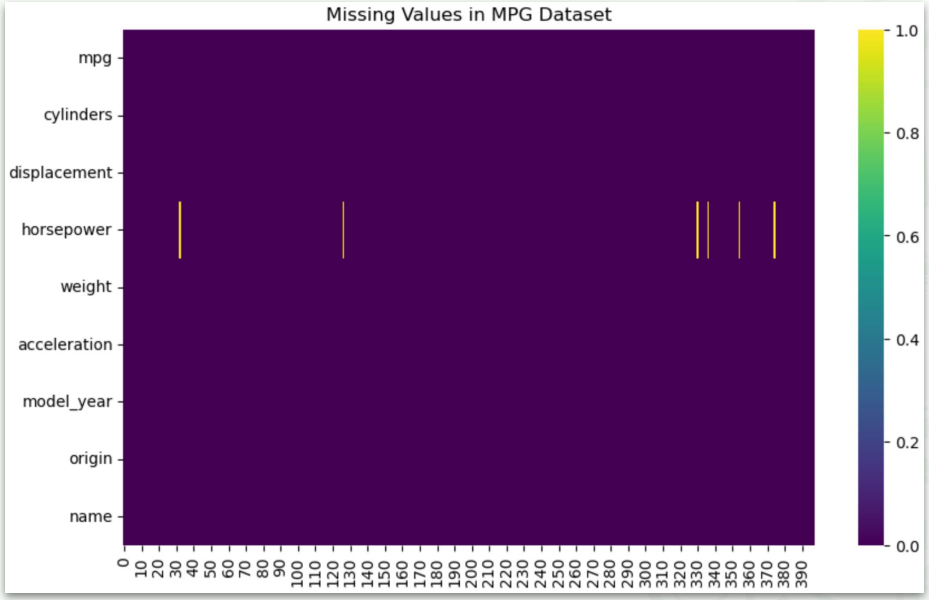
.describe()

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year
count	398.000000	398.000000	398.000000	392.000000	398.000000	398.000000	398.000000
mean	23.514573	5.454774	193.425879	104.469388	2970.424623	15.568090	76.010050
std	7.815984	1.701004	104.269838	38.491160	846.841774	2.757689	3.697627
min	9.000000	3.000000	68.000000	46.000000	1613.000000	8.000000	70.000000
25%	17.500000	4.000000	104.250000	75.000000	2223.750000	13.825000	73.000000
50%	23.000000	4.000000	148.500000	93.500000	2803.500000	15.500000	76.000000
75%	29.000000	8.000000	262.000000	126.000000	3608.000000	17.175000	79.000000
max	46.600000	8.000000	455.000000	230.000000	5140.000000	24.800000	82.000000

.info()

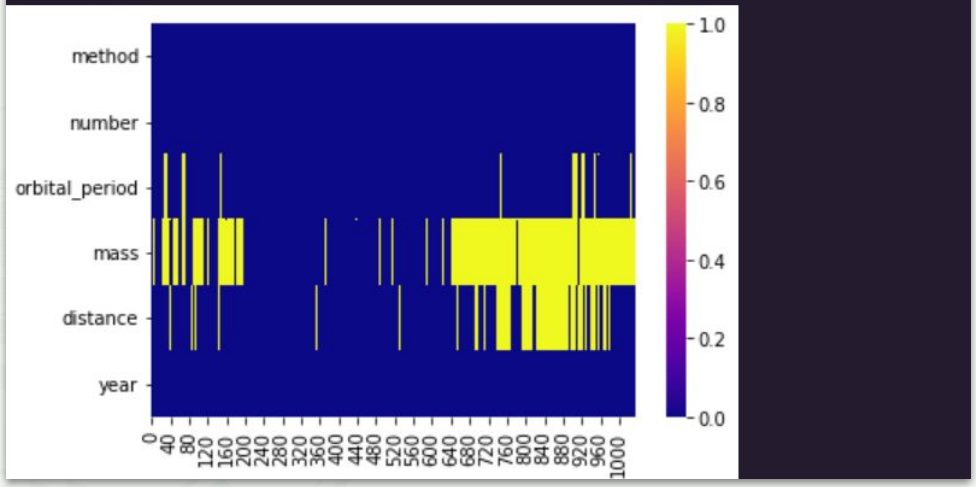
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   mpg           398 non-null    float64
 1   cylinders     398 non-null    int64  
 2   displacement  398 non-null    float64
 3   horsepower    392 non-null    float64
 4   weight        398 non-null    int64  
 5   acceleration  398 non-null    float64
 6   model_year    398 non-null    int64  
 7   origin        398 non-null    object 
 8   name          398 non-null    object 
dtypes: float64(4), int64(3), object(2)
memory usage: 28.1+ KB
```

Visualize Missing Data: Part of the EDA Process



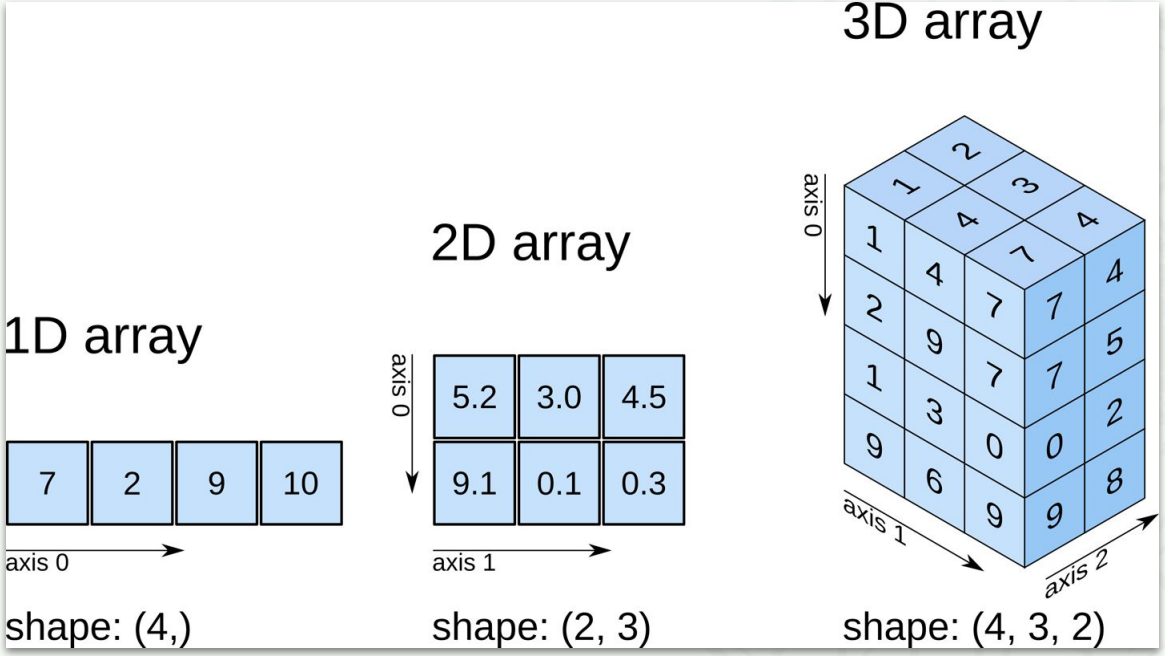
Seaborn: mpg

```
1 sns.heatmap(df_pl.isna().transpose(),cmap="plasma");
```



Seaborn: planets

NumPy Arrays



NumPy provides the `ndarray` object, which allows you to create arrays of any dimension.

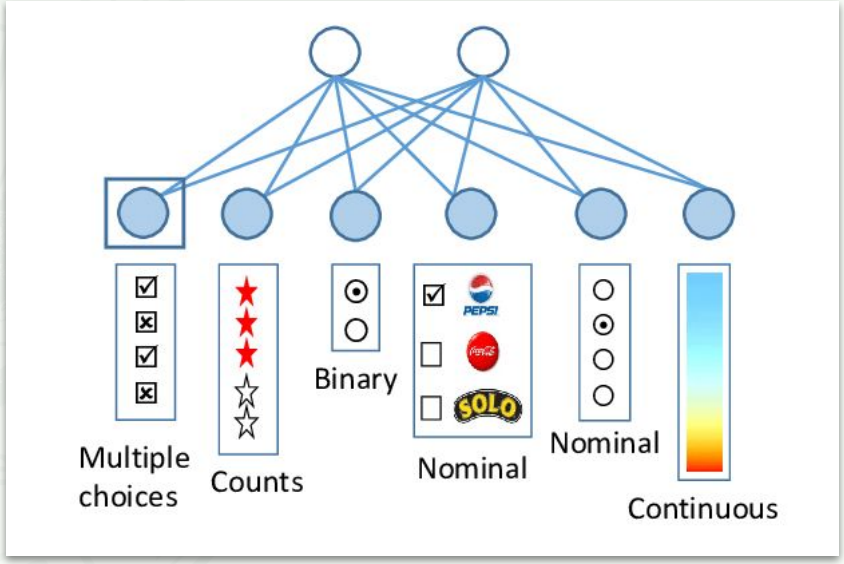
For many tasks, this is a very convenient way to organize data. Many libraries will assume your data is organized this way.

Are there other ways to organize data?

NumPy Won't Always Work

Data is a collection of discrete objects:

- numbers
- words
- facts
- objects
- measurements
- observations
- descriptions
- and so on....



How would we invent a simple “data matrix” structure in Python?

Probably: *with a dictionary*.

state	color	food	age	height	score
NY	blue	Steak	30	165	4.6
TX	green	Lamb	2	70	8.3
FL	red	Mango	12	120	9.0
AL	white	Apple	4	80	3.3
AK	gray	Cheese	32	180	1.8
TX	black	Melon	33	172	9.5
TX	red	Beans	69	150	2.2

```
data_dict = {'state': ['NY', 'TX', 'FL', 'AL', 'AK', 'TX', 'TX'],
             'color': ["blue", "green", "red", "white", "gray", "black", "red"],
             'food': ["Steak", "Lamb", "Mango", "Apple", "Cheese", "Melon", "Beans"],
             "age": [30, 2, 12, 4, 32, 33, 69],
             "height": [165, 70, 120, 80, 180, 172, 150],
             "score": [4.6, 8.3, 9.0, 3.3, 1.8, 9.5, 2.2]}
```

```
data_dict
```

```
{'state': ['NY', 'TX', 'FL', 'AL', 'AK', 'TX', 'TX'],
 'color': ['blue', 'green', 'red', 'white', 'gray', 'black', 'red'],
 'food': ['Steak', 'Lamb', 'Mango', 'Apple', 'Cheese', 'Melon', 'Beans'],
 'age': [30, 2, 12, 4, 32, 33, 69],
 'height': [165, 70, 120, 80, 180, 172, 150],
 'score': [4.6, 8.3, 9.0, 3.3, 1.8, 9.5, 2.2]}
```

We Can Go Far Beyond Putting Data Into Dictionaries

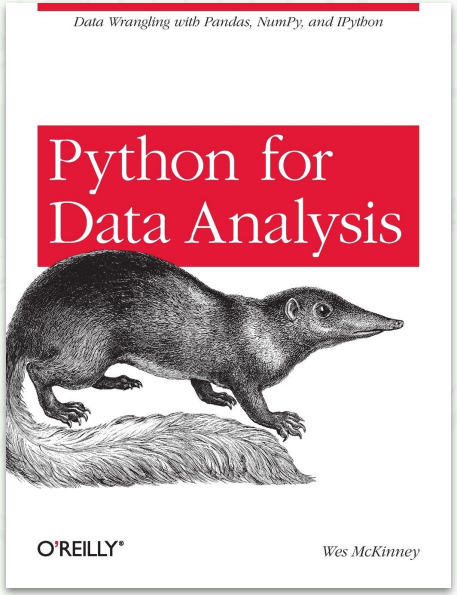
AQR CAPITAL MANAGEMENT

hedge fund

Wes McKinney

python™

Panel **Da**ta



pandas

Pandas: What is it?

Pandas is a software library for data science.

In particular, Pandas allows for data analysis and manipulations (e.g., munging and wrangling).

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Just as NumPy brings the array object into Python, Pandas brings in its own object, the

dataframe.

And, as with all other objects, there are a vast supply of methods.

Pandas Basics

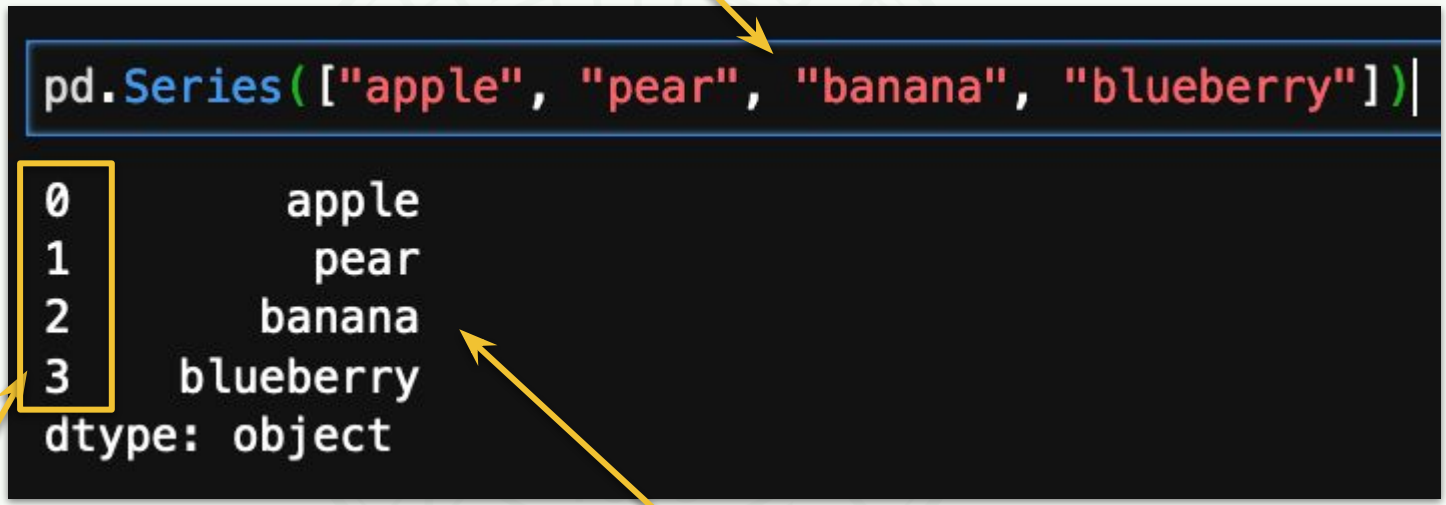
- Pandas is built from NumPy.
  - you don’t need to import NumPy,
  - but, you need NumPy on your computer (a good reason to use a distro like Anaconda)
- The standard way to import is:
  - `import pandas as pd`

Fundamental Pandas Objects

1. Index
2. Series
3. DataFrame
4. Panel Deprecated as of Pandas 0.25.0

Series

here, making the `Series` from a normal Python list



`Series` has added an `Index`

`Series` displays as a column

Series Attributes (Have No Parentheses)

```
my_series.shape
```

```
(4,)
```

```
my_series.head
```

```
<bound method NDFrame.head of 0        apple
1         pear
2       banana
3    blueberry
dtype: object>
```

```
my_series.values
```

```
array(['apple', 'pear', 'banana', 'blueberry'], dtype=object)
```

```
my_series.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

Attributes

`T`	Return the transpose, which is by definition self.
`array`	The ExtensionArray of the data backing this Series or Index.
`at`	Access a single value for a row/column label pair.
`attrs`	Dictionary of global attributes of this dataset.
`axes`	Return a list of the row axis labels.
`dtype`	Return the dtype object of the underlying data.
`dtypes`	Return the dtype object of the underlying data.
`flags`	Get the properties associated with this pandas object.
`hasnans`	Return True if there are any NaNs.
`iat`	Access a single value for a row/column pair by integer position.
`iloc`	Purely integer-location based indexing for selection by position.
`index`	The index (axis labels) of the Series.

Series Methods (Have Parentheses)

```
[34]: my_series.describe()

[34]: count          4
      unique         4
      top        apple
      freq           1
      dtype: object
```

Note: these are descriptive statistics of the *columns*.

Methods

`abs`()	Return a Series/DataFrame with absolute numeric value of each element.
`add`(other[, level, fill_value, axis])	Return Addition of series and other, element-wise (binary operator *add*).
`add_prefix`(prefix)	Prefix labels with string *prefix*.
`add_suffix`(suffix)	Suffix labels with string *suffix*.
`agg`([func, axis])	Aggregate using one or more operations over the specified axis.
`aggregate`([func, axis])	Aggregate using one or more operations over the specified axis.
`align`(other[, join, axis, level, copy, ...])	Align two objects on their axes with the specified join method.
`all`([axis, bool_only, skipna, level])	Return whether all elements are True, potentially over an axis.
`any`([axis, bool_only, skipna, level])	Return whether any element is True, potentially over an axis.
`append`(to_append[, ignore_index, ...])	(DEPRECATED) Concatenate two or more Series.
`apply`(func[, convert_dtype, args])	Invoke function on values of Series.
`argmax`([axis, skipna])	Return int position of the largest value in the Series.
`argmin`([axis, skipna])	Return int position of the smallest value in the Series.
`argsort`([axis, kind, order])	Return the integer indices that would sort the Series values.

The `Index` (noun!) allows you to index (verb!)

```
my_series

0         apple
1          pear
2        banana
3     blueberry
dtype: object
```

```
my_series.iloc[1]
```

```
'pear'
```

```
my_series.iloc[2]
```

```
'banana'
```

`loc` and `iloc`

access by name/label

access by integer/position

Adding Labels To `Index`, Using `loc`

```
my_series = pd.Series(["apple", "pear", "banana", "blueberry"], index=["1st", "2nd", "3rd", "4th"])
```

```
my_series
```

```
1st         apple
2nd          pear
3rd        banana
4th     blueberry
dtype: object
```

```
my_series.loc["4th"]
```

```
'blueberry'
```

Multi-Index

You can have multiple indices.

```
my_series = pd.Series(["apple", "pear", "banana", "blueberry"],
                      index=[["1st", "2nd", "3rd", "4th"],[0,1,2,3]])
```

```
my_series
```

```
1st  0        apple
2nd  1         pear
3rd  2       banana
4th  3    blueberry
dtype: object
```

Hierarchical Indexing

You can have *hierarchical* indices.

```
my_series = pd.Series(["apple", "pear", "banana", "blueberry"],
                      index=[["1st", "1st", "2nd", "2nd"],[0,1,0,1]])
```

```
my_series
```

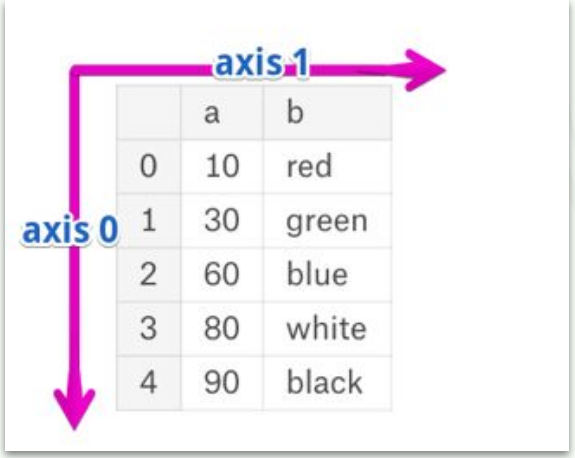
```
1st  0        apple
     1         pear
2nd  0       banana
     1    blueberry
dtype: object
```

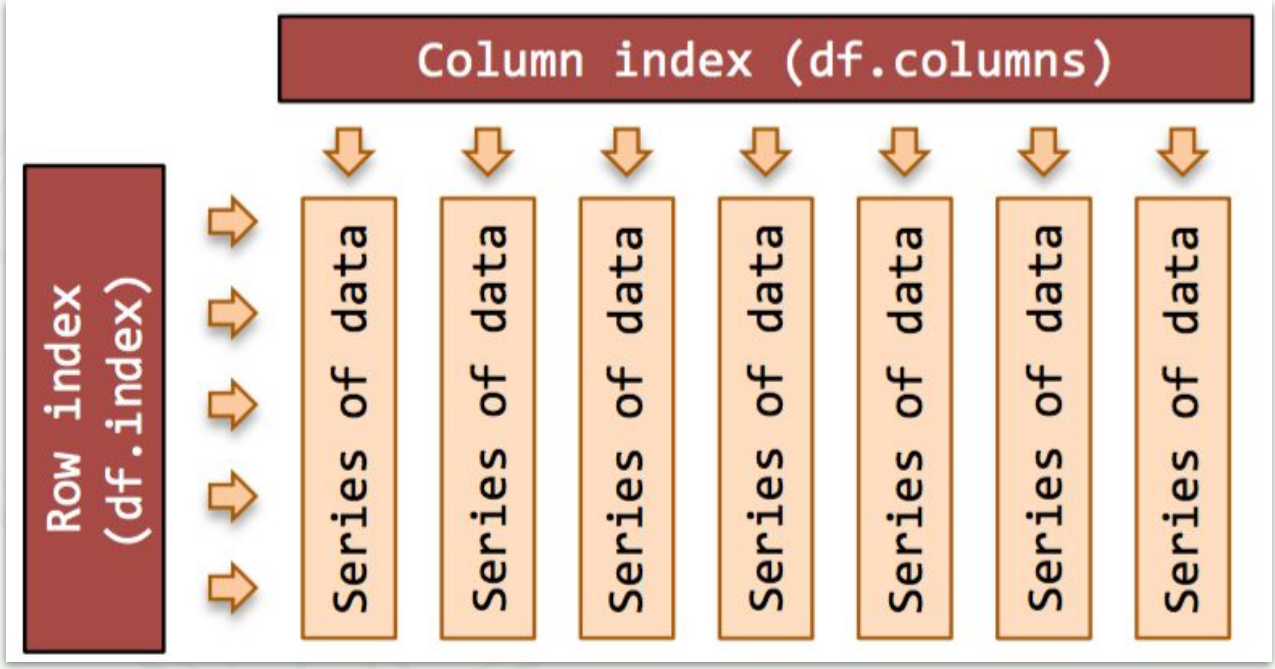
```
my_series.loc["1st"]
```

```
0     apple
1      pear
dtype: object
```

DataFrames

`DataFrames`, which are far more commonly used, can be thought of as a stacked `Series` with a column index.





DataFrame from Series

```
series_1 = pd.Series(range(5))
series_2 = pd.Series(["a", "b", "car", "cat", "cab"])
```

```
series_1
```

```
0    0
1    1
2    2
3    3
4    4
dtype: int64
```

```
series_2
```

```
0      a
1      b
2    car
3    cat
4    cab
dtype: object
```

```
my_frame = {"title 1": series_1, "title 2": series_2}
```

```
my_first_df = pd.DataFrame(my_frame)
```

```
my_first_df
```

	title 1	title 2
0	0	a
1	1	b
2	2	car
3	3	cat
4	4	cab

`DataFrame` from a dictionary

pandas.DataFrame.from_dict

classmethod DataFrame.**from_dict**(*data*, *orient='columns'*, *dtype=None*, *columns=None*)

Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specificatio

Parameters: **data** : *dict*

Of the form {field : array-like} or {field : dict}.

orient : *{'columns', 'index', 'tight'}, default 'columns'*

The "orientation" of the data. If the keys of the passed dict should be the the resulting DataFrame, pass 'columns' (default). Otherwise if the keys rows, pass 'index'. If 'tight', assume a dict with keys ['index', 'columns', 'd 'index_names', 'column_names'].

New in version 1.4.0: 'tight' as an allowed value for the `orient` arg

dtype : *dtype, default None*

Data type to force, otherwise infer.

columns : *list, default None*

Column labels to use when `orient='index'`. Raises a ValueError if used with `orient='columns'` or `orient='tight'`.

Returns: **DataFrame**

```
>>> import pandas as pd
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> df = pd.DataFrame.from_dict(data)
>>> df
   col_1 col_2
0      3     a
1      2     b
2      1     c
3      0     d
>>> df['col_1']
0    3
1    2
2    1
3    0
Name: col_1, dtype: int64
```

Note that a `Series` was returned.

`DataFrame` from a file

pandas.read_csv

pandas.read_csv(*filepath_or_buffer, sep=NoDefault.no_default, delimiter=None, header='infer', names=NoDefault.no_default, index_col=None, usecols=None, squeeze=None, prefix=NoDefault.no_default, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=None, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None, compression='infer', thousands=None, decimal='.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, encoding_errors='strict', dialect=None, error_bad_lines=None, warn_bad_lines=None, on_bad_lines=None, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None, storage_options=None*) [sourc

Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

Parameters: **filepath_or_buffer** : *str, path object or file-like object*

Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handle (e.g. via builtin `open` function) or `StringIO`.

sep : *str, default ','*

Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and

pandas.read_json

pandas.read_json(*path_or_buf=None, orient=None, typ='frame', dtype=None, convert_axes=None, convert_dates=True, keep_default_dates=True, numpy=False, precise_float=False, date_unit=None, encoding=None, encoding_errors='strict', lines=False, chunksize=None, compression='infer', nrows=None, storage_options=None*) [source]

Convert a JSON string to pandas object.

Parameters: **path_or_buf** : *a valid JSON str, path object or file-like object*

Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, file://localhost/path/to/table.json.

If you want to pass in a path object, panda

By file-like object, we refer to objects with (e.g. via builtin `open` function) or `StringIO`.

orient : *str*

Indication of expected JSON string format produced by `to_json()` with a correspondi orients is:

- `'split'` : dict like `{index -> [index], [values]}`
- `'records'` : list like `[{column -> value}`
- `'index'` : dict like `{index -> {column ->`
- `'columns'` : dict like `{column -> {index`
- `'values'` : just the values array

pandas.read_excel

pandas.read_excel(*io, sheet_name=0, header=0, names=None, index_col=None, usecols=None, squeeze=None, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skiprows=None, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, parse_dates=False, date_parser=None, thousands=None, decimal='.', comment=None, skipfooter=0, convert_float=None, mangle_dupe_cols=True, storage_options=None*) [source]

Read an Excel file into a pandas DataFrame.

Supports *xls*, *xlsx*, *xlsm*, *xlsb*, *odf*, *ods* and *odt* file extensions read from a local filesystem or URL. Supports an option to read a single sheet or a list of sheets.

Parameters: **io** : *str, bytes, ExcelFile, xlrd.Book, path object, or file-like object*

Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.xlsx.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handle (e.g. via builtin `open` function) or `StringIO`.

sheet_name : *str, int, list, or None, default 0*

Strings are used for sheet names. Integers are used in zero-indexed sheet positions (chart sheets do not count as a sheet position). Lists of strings/integers are used to request multiple sheets. Specify None to get all worksheets.

Available cases:

Mixing Pandas and NumPy

pandas.Series.to_numpy

`Series.to_numpy(dtype=None, copy=False, na_value=NoDefault.no_default, **kwargs)` [source]

A NumPy ndarray representing the values in this Series or Index.

Parameters:

dtype : *str or numpy.dtype, optional*

The dtype to pass to `numpy.asarray()`.

copy : *bool, default False*

Whether to ensure that the returned value is not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

na_value : *Any, optional*

The value to use for missing values. The default value depends on *dtype* and the type of the array.

> *New in version 1.0.0.*

****kwargs**

Additional keywords passed through to the `to_numpy` method of the underlying array (for extension arrays).

> *New in version 1.0.0.*

pandas.DataFrame.to_numpy

`DataFrame.to_numpy(dtype=None, copy=False, na_value=NoDefault.no_default)` [source]

Convert the DataFrame to a NumPy array.

By default, the dtype of the returned array will be the common NumPy dtype of all types in the DataFrame. For example, if the dtypes are `float16` and `float32`, the results dtype will be `float32`. This may require copying data and coercing values, which may be expensive.

Parameters:

dtype : *str or numpy.dtype, optional*

The dtype to pass to `numpy.asarray()`.

copy : *bool, default False*

Whether to ensure that the returned value is not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

na_value : *Any, optional*

The value to use for missing values. The default value depends on *dtype* and the dtypes of the DataFrame columns.

> *New in version 1.1.0.*

Returns: numpy.ndarray