# Numpy

To access the array underlying a DataFrame or Series, use the `to_numpy` method.

**Note**: `to_numpy()` not return the copy of series, the change on the `.to_numpy` will change the original series.

`np.random.random(N)` returns an array containing N numbers selected uniformly at random from the interval [0, 1).

`np.clip(series/number, lower_bound, upper_bound)`

```
np.count_nonzero() # count non zero
np.percentile(arr, 95) # Find the 95% percentile of arr
np.random.choice(['H', 'T'], p=[0.5, 0.5], size=114)
# Faster simulation; 2D Array
np.random.choice(outcome, p_for_each, size=(number_repetition, num_each_trial))
np.random.multinomial(10, [0.1, 0.2, 0.3])
np.random.permutation(ser/arr)
```

# Series

```
ser.plot(kind=, density=, bins=, title=)
ser.apply(func) # Apply funciton or lambda to series
ser.to_numpy() # 改变array也会改变series
ser.astype(int) # Change type of series
# Note: some have special characters that can't typechange directly
ser.unique()
ser.nunique() # number of unique values of this column
ser.value_counts() # count the number of each unique values
ser.describe() # describtion about mean, max, min, std, ect; also work for df
ser.str.split().str[0] #accessing every 1st element strip
ser.replace({dict}) # replace with dictionary
ser.str.zfill(len) #adds zeros to the start until total reaches length
ser.isna() #element-wise
ser.dropna() #returns a new Series with all null entries removed
ser.rename(new_name) # change the name of series
ser.rename(lambda x: x ** 2)  # function, changes labels
ser.rename({1: 3, 2: 5})  # mapping, changes labels or index
ser.diff() # Difference with previous
ser.isin([values]) #if elements in Series are contained in values
ser.index # Get all index of series as index array
ser.index[index_num] # The index value of that index position
ser.loc[index] # find the value of that index
ser.iloc[num] # find the value of the num th row (Start from 0)
```

## Pandas

### Initialize Series and DataFrame

```
# Create Series with dictionary
pd.Series({'a': 10, 'b': 23, 'c': 45, 'd': 53, 'e': 87}, name='people') # name is optional
# Create DataFrame with dictionary
column_dict = {'Name': ['Granger, Hermione', 'Potter, Harry', 'Weasley, Ron', 'Longbottom, Neville'],
               'PID': ['A13245986', 'A17645384', 'A32438694', 'A52342436'],
               'LVL': [1, 1, 1, 1]}
enrollments = pd.DataFrame(column_dict)
# Initialize an empty DataFrame with N rows
new_df = pd.DataFrame(index=range(N))
```

`df.index` and `df.columns`

Axis 0 refers to the index; 纵向压缩 `.sum(axis=0)`

Axis 1 refers to the column 横向压缩 `.sum(axis=1)`

### Get rows/columns

```
# Returns a Series
enrollments['Name']
# Returns a DataFrame; Select multiple columns
enrollments[['Name', 'PID']]
# Get multiple rows
enrollments.loc[[1, 2]]
# Rows where Name includes 'on'
enrollments.loc[enrollments['Name'].str.contains('on')]
df.loc[<row selector>, <column selector>] # select the rows and columns as the same time
df.loc[[row1, row2]] # Select multiple rows
df.loc[:, :] # select all rows and columns
df.loc[name1:name2, col1: col2] # select rows and columns within the names and cols, inclusive
df.iloc[num_row1:num_row2, num_col1:num_col2] # Can't put specific name, exclusive
df.loc[index_of_insert] = [row to append] # Add the row to the last row by loc
```

横纵坐标都要放进去要用`.loc`

`[]`放一个坐标默认get column (need column name), `.loc[]`放一个坐标默认row (row index)

`df[]`: Boolean arrays always select rows by default.

`.iloc[]`Without know the label of row, 在不知道index的情况下，去找某个column在第几排的值

`df.loc[row, column]` ; df[input]是取符合的row index，不是columns

`.iloc[-1]` returns the last row of table

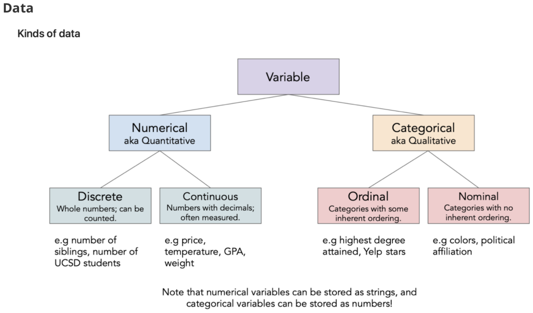
The dtypes attribute (of both Series and DataFrames) describes the data type of each column.

The `to_numpy` method, when used on a Series, returns an array in which all values are of the data type specified by dtypes .

The `to_numpy` method, when used on a DataFrame, returns a multi-dimensional array of type object , unless all columns in the DataFrame are homogenous.

The `head / tail` methods return the first/last few rows (the default is 5).

### Messy Data




#### Replace

```
.replace({'original': 'replace', 'original2':'replace2'}) # use dictionary to replace multiple columns
df[column].replace() # works for str column
df = df.replace({column: dict_for_replace})
df[column].str.contians() # if the method is a string method, need `.str`. e.g: `index(), isdecimal(),
isnumeric(), islower(), split(), strip(), upper()`.
```

```
pd.to_numeric(students['DSC 80 Final Grade'], errors='coerce')
```

Careful when using `errors='coerce'`, some information may be lost when using it

Use `.str` to access the str attributes of series.

### Unfaithful data and outlier

Unfaithful data are data that don't accurately represent the data generating process.

Outliers are "unusual" observations, unlike the rest of the data. They may be real, or they may be unfaithful.

`.describe()`: see basic numerical information about a Series/DataFrame.

`.info()`: see data types and the number of missing values in a DataFrame.

`.value_counts()` : see the distribution of a categorical variable.

`.plot(kind='hist')` : plot the distribution of a numerical variable.

#### NaN

`type(np.NaN)` is float, pay attention to type coercion!

`np.NaN == np.NaN` # but there are not equal -> False

`isnull()` or `isna()` to find the np.NaN or None

The result of any comparison (==, !=, <, >) with np.NaN is False.

```
df.size or ser.size # size will count null; # df.size will give back all 格子数量
df.count() or ser.count() # count will not count null
df.mean() or ser.mean() # Mean will not count null
df[column].count() / df[column].size # Calculate the proportion of non NaN of this column
```

```
df.dropna() # In place # Return Null; drop rows contains at least one null values
df.dropna(how='any') # drop all rows contians np.NaN
df.dropna(how='all') # .dropna() not mean you drop all of your rows containing NaN
df.dropna(axis=1) # drops columns contains at least one null values
df.dropna(subset=['A', 'B']) # Only consider column A and B
df.fillna(val) # fills null entries with the value val
df.fillna(dict) # fills null entries using a dictionary dict of column/row values.
df.fillna(method='bfill') # fill null entries using neighboring non-null entries, back fill
df.fillna(method='ffill') # forward fill (pull up one to down)
df.fillna({col: val})
# Another way of doing the same thing #### The lambda takes in each column
df.apply(lambda x: x.fillna(x.mean()), axis=0)
```

### Groupby (split, apply, and combine)

The groupby method can often produce results using just a single pass over the data, updating the sum, mean, count, min, or other aggregate for each group along the way.

**`.apply()`**

It accepts a group as a DataFrame/Series, and can return a DataFrame, Series, or scalar

可以apply单个column，也可以调用multiple columns; Cross column

**`.applymap()`**

Elementwise

**`.transform()`**

A transformation returns a DataFrame or Series of the same size; result will be the same length as original DataFrame

只能transform单个column

```
penguins.groupby('species')['body_mass_g'].transform(lambda ser: ser - ser.mean())
```

**`.agg()`**:

apply some function to each group, and combine the results; result will be the length of the number of groups

```
# can use multiple functions on a column at the same time
penguins.groupby('species').aggregate({'bill_length_mm': 'max', 'island': ['nunique', 'max']})
```

**`.filter()`**

keep only the groups that satisfy a particular condition

```
penguins.groupby('species').filter(lambda df: df['bill_length_mm'].mean() > 39)
```

Groupby with multiple columns

```
double_group = penguins.groupby(['species', 'island'])
penguins.groupby(['species', 'island'], as_index=False).mean() #Shortcut to use reset index
```

### Pivot table

pivot_table = groupby + pivot

```
df.pivot_table(index=index_col, columns=columns_col, values=values_col, aggfunc=func).fillna(0) # As float
# Pivot
moves = pd.DataFrame([ [1, 1, 'O'],
     [2, 1, 'X'],
     [2, 2, 'X'],
     [2, 3, 'O'],
     [3, 1, 'O'],
     [3, 3, 'X']
], columns=['i', 'j', 'move']) moves
moves.pivot(index='i', columns='j', values='move').fillna('')
```

The pivot method only reshapes a DataFrame. It does not change any of the values in it (i.e. aggfunc doesn't work with pivot ).

Find the number of penguins per island and species.

```
penguins.pivot_table(index='island',
                     columns='species',
                     values='bill_length_mm',
                     aggfunc='count')
```

| species | Adelie | Chinstrap | Gentoo |
|---|---|---|---|
| **island** | | | |
| **Biscoe** | 44.0 | NaN | 119.0 |
| **Dream** | 55.0 | 68.0 | NaN |
| **Torgersen** | 47.0 | NaN | NaN |

### Simpson's paradox

Simpson's paradox occurs when grouped data and ungrouped data show opposing trends.

It often happens because there is a hidden factor (i.e. a confounder) within the data that influences results.

## Conbination of dataFrame

### `pd.concat()` Row-wise combination of data

默认竖着concat

```
pd.concat([section_A, section_B], ignore_index=True) # Fix the index
pd.concat([section_A, section_B], keys=['Section A', 'Section B']) # keep track of which original DataFrame each
row came from
pd.concat([exams, assignments], axis=1) # columns next to columns; default is axis=0
```

If we concatenate two DataFrames that don't share the same column names, NaN s are added in the columns that aren't shared.

`os.listdir(dirname)`returns a list of the names of the files in the folder

`pd.concat` only looks at the index when combining rows, not at any other columns.

## Merge

If join keys are not specified, all shared columns between the two DataFrames are used by default.

**Pay attention to specify which column to merge on, otherwise will merge on all columns**

```
temps.merge(countries, how='outer')
# merge is also a pandas function
pd.merge(temps, countries, how='outer')
exams.merge(overall, left_on='Name', right_on='Student')
exams.merge(overall, left_on='Name', right_on='Student', suffixes=('_Exam', '_Overall'))
```

**Inner**: keep only matching keys (**intersection**).

**Outer**: keeps all keys in both DataFrames (**union**).

**Left**: keep all keys in the left DataFrame, whether or not they are in the right DataFrame.

**Right**: keep all keys in the right DataFrame, whether or not they are in the left DataFrame.

**One-to-one joins:**

Neither the left DataFrame nor the right DataFrame contained any duplicates in the join key.

**Many-to-one joins:**

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

The resulting DataFrame will preserve those duplicate entries as appropriate.

**Many-to-many joins:**

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

# Hypothesis Testing

Note that we are very careful in saying that we either **reject the null** or **fail to reject the null**.

The p-value is the probability, under the assumption the null hypothesis is true, of observing a test statistic **equal to our observed statistic, or more extreme in the direction of the alternative hypothesis**.

## Difference

The signed difference between the mean/median of two groups; Alternative 有方向

The unsigned (absolute) difference between the mean/median of two groups; 两个distribution有什么区别

## TVD

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

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

The **Total Variation Distance (TVD)** of **two categorical distributions** is **the sum of the absolute diferences of their proportions, all divided by 2**.

```
np.sum(np.abs(dist1-dist2)) / 2
```

Note: Total Variation Distance is only use for comparing two categorical distribution

```
np.random.multinomial(10, [0.5, 0.5]) # 10 times with [0.5, 0.5] possibility
np.random.multinomial(totol_pop, [.., .., ..], size=(trial))
```

If the two distributions are **quantitative (numerical)**, we use as our test statistic the **difference in group means or medians**.

If the two distributions are **qualitative (categorical)**, we use as our test statistic the **total variation distance (TVD)**.

# Permutation Test

Given two observed samples, are they fundamentally different, or could they have been generated by the same process?

In a permutation test, we decide whether two fixed random samples come from the same distribution.

In a permutation test, we generate new data by **shuffling group labels**.

**To test whether two distributions come from the same underlying population distribution.**

To create a permutation, either set `n=df.shape[0]` or `frac=1`. `smoking_and_birthweight.sample(frac=1)`

frac from 0 to 1, give the size of the proportion of the dataFrame

E.x:

Null hypothesis: In the population, birth weights of smokers and non-smokers have the same distribution. The difference we saw was due to random chance.

Alternative hypothesis: In the population, babies born to smokers have lower birth weights, on average.

**Permutation Test Sample**

**Null hypothesis**: the two sample are from same distribution, the difference is due to random chance

**Alternative hypothesis**: the two sample are from different distribution. or the one is lower

```
# Calculate TVD
def tvd_of_groups(df):
    cnts = df.pivot_table(index=distribution, columns=category, aggfunc='size')
    distr = cnts / cnts.sum()   # Normalized
    return distr.diff(axis=1).iloc[:, -1].abs().sum() / 2  # TVD
```

```
# Permutation Test Sample Code
# Observed test statistic
observed_difference = (
    df
    .groupby('groupby_column')['want_to_shuffled'] .mean()
    .diff()
    .iloc[-1]
)
# Simulation
n_repetitions = 500
differences = []
for _ in range(n_repetitions):
    # Step 1: Shuffle the weights
    shuffled_column = (
        df['want_to_shuffled']
        .sample(frac=1)
        .reset_index(drop=True) # Be sure to reset the index!
    )
    # Step 2: Put them in a DataFrame
    shuffled = (
        df
        .assign(**{'Shuffled column': shuffled_column})
    )
    # Step 3: Compute the test statistic
    group_means = ( shuffled
        .groupby('groupby_column').mean()
        .loc[:, 'Shuffled column']
    )
    difference = group_means.diff().iloc[-1]
    # Step 4: Store the result
    differences.append(difference)
# Calculate p-value, if it tests the whether they from same distribution, diff should be small
pval = (differences >= observed_difference).mean()
# Reject the null if pval is very small
```

### The Kolmogorov-Smirnov test statistic

The K-S test statistic measures the similarity between two distributions.

t is defined in terms of the cumulative distribution function (CDF) of a given distribution.

If *f*(*x*) is a distribution, then the CDF *F*(*x*) is the proportion of values in distribution *f* that are less than or equal to *x*.

The K-S statistic is roughly defined as the largest difference between two CDFs.

**Only use to test whether the two have the same distribution. (Often with graph)**

```
# Other pd method
pd.read_csv()
pd.to_numeric(series, errors='coerce')
df.describe() #count, mean, std, 5 number summary
df.shape or df.size #size: #row*#col
df.index or df.columns #row/index label (axis=0) & column label (axis=1)
df.rename(index={}, columns={})
df.sum() #total; df.sum(axis=0) #sum each column; df.sum(axis=1) #sum each row
df['col'].str.contains()
df.dtypes #显示每个column's data type as a series
df.mean() #mean of each col (如果能取mean)
df.mean(axis=1) #mean of each rows
df.drop_duplicates(subset=[col]) #保留该col第一次出现的值
df.assign(**{'A B': some_lst}) #keyword argument使col name能有空格
df.reset_index(drop=True)/sort_index()/set_index() # reset an index of increasing integers
df.value_counts(normalize=True).to_frame() # normalize the data # to_frame(): to DataFrame
df.groupby(key).groups #returns a dictionary that the keys are group names and values are lists of row
labels
df.groupby(key).get_group(key) # returns a df with only the values for the given key
df.groupby().aggregate([list of functions])
df.groupby().aggregate({'A': 'max', 'B: 'nunique'})
df.groupby().transform(lambda ser: ser - ser.mean())
df.groupby().filter(lambda df: df['b'].mean() > 39) # 保留符合条件的group
df.groupby([multiple cols]) # MultiIndex用df.loc[('A', 'B')] access
df.transpose()/df.T
pd.concat([df1, df2], ignore_index=True, keys=['A', 'B']) #df1.concat(df2) vertically
pd.concat([df1, df2], axis=1) #horizontally match index
df.sample() or df.sample(n) # sample 1 or n rows
df.sample(frac=).reset_index(drop=True) #shuffle, frac = 1
```

## Datetime

`pd.Timestamp()` is the pandas equivalent of datetime .

`pd.to_datetime()` or `x.time()` converts strings to pd.Timestamp objects.

```
pd.Timestamp(year=1998, month=11, day=26)
final_start = pd.to_datetime('June 4th, 2022, 11:30AM')
# Other method
datetime.datetime.now() # The time for now
datetime.timedelta(days=3, hours=5) #过了多少时间, measure durations
datetime.datetime.now().timestamp() #从1970.1.1到现在过了多少秒
pd.to_datetime('June 4th, 2022, 11:30AM') # return a Timestamp object, 可以加减
pd.Timestamp.year/dayofweek/day/hour/min/sec # time-related attributes
```

Subtracting timestamps yields `pd.Timedelta` objects.

If we create a Series of datetimes with `pd.to_datetime` , pandas stores them as yet *another* type: `np.datetime64`

# Missing Values

**Missing by design (MD)**
*Can I determine the missing value exactly by looking at the other columns?* 🤔

**Not missing at random (NMAR)**
*Is there a good reason why the missingness depends on the values themselves?* 🤔

**Missing at random (MAR)**
*Do other columns tell me anything about the likelihood that a value is missing?* 🤔

**Missing completely at random (MCAR)**
*The missingness must not depend on other columns or the values themselves.* 😄

- **Missing by design (MD):** Whether or not a value is missing depends entirely on the data in other columns. In other words, if we can always predict if a value will be missing given the other columns, the data is MD.
- **Not missing at random (NMAR, also called NI):** The chance that a value is missing **depends on the actual missing value**!
- **Missing at random (MAR):** The chance that a value is missing **depends on other columns**, but **not** the actual missing value itself.
- **Missing completely at random (MCAR):** The chance that a value is missing is **completely independent** of other columns and the actual missing value.

## Handle Missing Value

### Drop Missing value

If the data are **MCAR**, then dropping the missing values entirely **doesn't significantly change the data**.

**If the data are not MCAR, then dropping the missing values will introduce bias.** (MCAR is rare)

**Likewise Deletion**: Dropping entire **rows** that contain missnig values. `.dropna()`. (issue: will delete good data in other columns)

### Imputation: Filling in missing data with plausable values; try not to introduce bias

**Imputation with a single value: mean, median, mode. `.fillna(df[col].mean())` or median or mode**

When data are **MCAR** and you impute with the mean: mean **unbiased** and variance **decreased**

When data are **MAR**, mean imputation leads to **biased** estimates of the mean across groups. (biased towards one group.)

**Within-group (conditional) Mean Imputation: Filling in missing values based on the columns they depend on**

Then, if data **MAR**, the overall mean remains **unbiased** but the variance of the dataset is **reduced**. Correlations **increased**.

If the column with missing values were dependent on ***more than one*** **column**, use linear regression to predict the missing value.

The new means may be biased low or high according to the original **not missing values**.

```
means = df.groupby('c2').mean().to_dict() # For a column c1, conditional on a second column c2
imputed = df['c1'].apply(lambda x: means[x] if pd.isna(x) else x)
```

**Imputation with a single value, using a model: regression, kNN.**

**Probabilistic imputation by drawing from a distribution** (Random) of non-missing data. **Variance is preserved.**

If a value was never observed in the dataset, it will never be used to fill in a missing value. **Solution**: Create a histogram (with `np.histogram`) to bin the data, then sample from the histogram.

If data are **MCAR**, the resulting mean and variance are unbiased estimates of the true mean and variance.

Extending to the **MAR** case: draw from conditional empirical distributions.

## HTTP Hypertext Transfer Protocol

### The request -response model

- A **request** is made by the **client**. `GET` is used to request data **from** a specified resource.
- A **response** is returned by the **server**. `POST` is used to **send** data to the server. (send content back to the client in its response.)

### Making HTTP requests

**From Python, with the `requests` package.**

```
# GET via requests
import requests
resp = requests.get(url) #return a response object (e.g. <Response [200]>) # 200 means success
resp.text # string that contains the entire response (html) # type(resp.text) -> str
resp.request.url # give the URL link we accessed
resp.status_code # get the status code for this request
resp.ok #check if a request was successful
# If rate of requests is too high, slow down requests between each retry using `time.sleep`
resp.raise_for_status # raises an exception when the status code is not-ok.
# POST via requests
post_response = requests.post('https://httpbin.org/post', data={'name': 'King Triton'})
```

### HTTP status codes

The most common status code is `200` -> no issues. or error(e.g. `404`: page not found; `500`: internal server error.)

# JSON: JavaScript Object Notation

The two main file formats used for storing information on the internet are **HTML** and **JSON**.

### JSON data types

- string: anything inside double quotes.
- number: any number (no difference between ints and floats).
- boolean: `true` and `false`.
- array: anything wrapped in `[]`.
- null: JSON's empty value, denoted by `null`.
- object: a collection of key-value pairs (like dictionaries).
  - Keys must be strings, values can be anything (even other objects).

```
import json
```

```
f = open(os.path.join('data', 'family.json'), 'r')
family_tree = json.load(f)
```

```
family_tree
```

```
{'name': 'Grandma',
 'age': 94,
 'children': [{'name': 'Dad',
   'age': 60,
   'children': [{'name': 'Me', 'age': 23}, {'name': 'Brother', 'age': 21}]},
  {'name': 'My Aunt',
   'children': [{'name': 'Cousin 1', 'age': 34},
    {'name': 'Cousin 2',
     'age': 36,
     'children': [{'name': 'Cousin 2 Jr.', 'age': 2}]}]}]}
```

```
family_tree['children'][0]['children'][0]['age']
```

```
23
```

**eval: stands for "evaluate": `eval('4 + 5')` -> 9**

不能用于json: This happened because `eval` evaluates all parts of the input string as if it were Python code.

```
import json
json.load(file) or json.loads(str) # loads a JSON file from a file or string
requests.get(url).json() # display in json
```

## APIs and web scraping

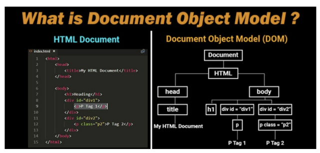
### API requests: just GET/POST requests to a specially maintained URL.

```
r = requests.get('https://pokeapi.co/api/v2/pokemon/squirtle') # <Response [200]>
r.content # Get the content for this requested URL
# We can extract the JSON from this request with the json method (or by passing r.text to json.loads).
r.json()
```

### Scraping

Programmatically "browsing" the web, downloading the source code (HTML) of pages; May not be able to scrap some websites.

**`robots.txt` : this file in their root directory, which contains a policy that allows or disallows automatic access to their site.**




## HTML (HyperText Markup Language)

### The anatomy of HTML documents

- **HTML document**: The totality of markup that makes up a webpage.
- **Document Object Model (DOM)**: The internal representation of a HTML document as a hierarchical **tree** structure.
- **HTML element**: An object in the DOM, such as a paragraph, header, or title.
- **HTML tags**: Markers that denote the **start** and **end** of an element, such as `<p>` and `</p>`.

### Useful tags to know

| Element | Description | Element | Description | Element | Description |
|---|---|---|---|---|---|
| `<html>` | the document | `<h1>, <h2>, ...` | header(s) | `<p>` | a paragraph |
| `<div>` | a logical division of the document | `<a>` | an anchor (hyper-link) | `<img>` | an image |
| `<span>` | an *in-line* logical division | `<head>` | the header | `<body>` | the body |

Tags can have **attributes**, which further specify how to display information on a webpage.

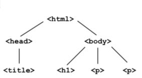
The `<div>` element is often used as a container for other HTML elements to style them with CSS or to perform operations involving them using JavaScript.

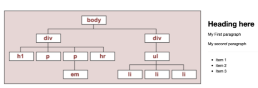
### Document Trees

```
!cat data/lec15_ex1.html
```

```
<html>
        <head>
                <title>Page title</title>
        </head>

        <body>
                <h1>This is a heading</h1>
                <p>This is a paragraph.</p>
                <p>This is another paragraph.</p>
        </body>
</html>
```




### Parsing HTML via Beautiful Soup: A `BeautifulSoup` object represents a node in the tree.

**Child nodes**

`soup.children` isn't another `BeautifulSoup` object, but rather something of the form `<list_iterator at 0x7f7b0ab8c370>`.

The `children` attribute returns an iterator so that it doesn't have to load the entire DOM tree in memory.

**Note**: 数child nodes时不要把</结束>数进去

The `soup.descendants` attribute traverses a `BeautifulSoup` tree using **depth-first traversal**.

**Finding elements in a tree and note attributes**

```
import bs4
soup = bs4.BeautifulSoup(html_string)
print(soup.text) # print out beautiful soup of HTML
soup.children #list iterator
soup.find(tag) #first instance of a tag
soup.find(tag).text # the text between the opening and closing tags.
soup.find(name=None, attrs={}, recursive=True, text=None, **kwargs) # General
soup.find_all(tag) #list of all instances of a tag
soup.find(tag).get(attri) # gets the value of a tag attribute
soup.find('div').attrs # You can access tags using attribute notation, too.
soup.find('div', attrs={'id': 'nav'}) # find the <div> element that has an id attribute equal to 'nav'.
soup.html.div.h1
soup.html.div.next_sibling.next_sibling.attrs
```

If you scraping a web page and never finishes and not rasie an error -> Have too many requests to the server in too short of a time, and you are being "timed out".

**Aside: f-strings in Python: convenient way to format strings.**

```
f'2 + 3 = {2 + 3}' # '2 + 3 = 5' # evaluate all things in { }
def make_greeting(name):
    return f"Hi {name}! 👋 Your name has {len(name)} characters, the first of which is {name[0]}."
make_greeting('Billy') # 'Hi Billy! 👋 Your name has 5 characters, the first of which is B.'
```

**Nested vs. flat data formats**

- **Nested** data formats, like HTML, JSON, and XML, allow us to represent hierarchical relationships between variables.
- **Flat** (i.e. tabular) data formats, like CSV, do not.

## Regular expression

**Regular Expression Reference**

| Operator | Description | Operator | Description |
|---|---|---|---|
| . | Matches any character except \n | ** | Escapes special characters |
| * | Matches preceding character/group zero or more times | \| | Matches expression on either side of expression |
| ? | Matches preceding character/group zero or one times | *? | non-greedy matching to * |
| + | Matches preceding character/group one or more times | +? | non-greedy matching to + |
| \d, \w, \s | character group of digits (0-9), alphanumerics (a-z, A-Z, 0-9, and underscore), or whitespace, respectively | \D, \W, \S | Inverse sets of \d, \w, \s |
| {m} | Matches preceding character/group exactly m times | ^ | Matches beginning of line |
| {m, n} | Matches preceding character/group at least m times and at most n times; if either m or n are omitted, set lower/upper bounds to 0 and ∞, respectively | $ | Matches end of the lin |
| {m,n}? | Matches the expression to its left **m** times, and ignores **n**. | [(+*)] | matches (, +, *, and ) |
| [ ] | Matching group used to match any of the specified characters or range (e.g. [abcde]) [a-e]) | ( ) | Matches the expression and groups it. |
| [^ ] | Invert matching group; e.g. [^a-c]matches all characters except a, b, c | .* | matches every possible string |

**Other special matching**

GROUPS

`( )` | Matches the expression inside the parentheses and groups it.

`(?)` | Inside parentheses like this, `?` acts as an extension notation. Its meaning depends on the character immediately to its right.

`(?PAB)` | Matches the expression AB, and it can be accessed with the group name.

`(?aiLmsux)` | Here, a, i, L, m, s, u, and x are flags:
- a — Matches ASCII only
- i — Ignore case
- L — Locale dependent
- m — Multi-line
- s — Matches all
- u — Matches unicode
- x — Verbose

`(?:A)` | Matches the expression as represented by A, but unlike `(?PAB)`, it cannot be retrieved afterwards.

`(?#...)` | A comment. Contents are for us to read, not for matching.

`A(?=B)` | Lookahead assertion. This matches the expression A only if it is followed by B.

`A(?!B)` | Negative lookahead assertion. This matches the expression A only if it is not followed by B.

`(?<=B)A` | Positive lookbehind assertion. This matches the expression A only if B is immediately to its left. This can only matched fixed length expressions.

`(?<!B)A` | Negative lookbehind assertion. This matches the expression A only if B is not immediately to its left. This can only matched fixed length expressions.

`(?P=name)` | Matches the expression matched by an earlier group named "name".

`(...)\1` | The number 1 corresponds to the first group to be matched. If we want to match more instances of the same expression, simply use its number instead of writing out the whole expression again. We can use from 1 up to 99 such groups and their corresponding numbers.

**Regular expression functions**

```
import re
re.findall(A, B) # Matches all instances of an expression A in a string B and returns them in a list.
re.search(A, B) # Matches the first instance of an expression A in a string B, and returns it as a rematch object.
re.split(A, B) # Split a string B into a list using the delimiter A.
re.sub(A, B, C) # Replace A with B in the string C.
```

## Bag of words: doesn't consider order; treat equally; not consider meaning

The bag of words model represents documents (e.g. job titles, sentences, essays) as vectors of word count.

**Cosine similarity and bag of words**: $\cos\theta = \frac{\vec{a}\cdot\vec{b}}{|\vec{a}||\vec{b}|}$: large -> two word vectors similar (nomalize length of vecotors)

**cosine distance** (the complement of cosine similarity): $\text{dist}(\vec{a},\vec{b}) = 1 - \cos\theta$. If $\text{dist}(\vec{a},\vec{b})$ is small, the two word vectors are similar.

## Parameters vs. hyperparameters

- A **parameter** defines the relationship between variables in a model. **We learn parameters from data**.
- A **hyperparameter** is a parameter that we get to choose **before our model is fit to the data**.

# Quantifying text data

## TF-IDF

### Term frequency

- The **term frequency** of a word (term) $t$ in a document $d$, denoted $\text{tf}(t, d)$ is the proportion of words in document $d$ that are equal to $t$.

$$\text{tf}(t, d) = \frac{\text{number of occurrences of } t \text{ in } d}{\text{total number of words in } d}$$

- If $\text{tf}(t,d)$ is **large**, then word $t$ occurs **often** in $d$.
- If $\text{tf}(t,d)$is **small**, then word $t$ does **not** occur often $d$.

**how often a word appears in a particular document**

### Inverse document frequency

- The **inverse document frequency** of a word $t$ in a set of documents $d_1, d_2, \ldots$ is

$$\text{idf}(t) = \log\left(\frac{\text{total number of documents}}{\text{number of documents in which } t \text{ appears}}\right)$$

Note: the inverse document frequency need to look at all documents. (total number of documents, not total number of words in documents)

- **how often a word appears across documents**
- If $\text{idf}(t)$ is **large**, then $t$ is **rarely** found in documents.
- If $\text{idf}(t)$ is **small**, then $t$ is **commonly** found in documents.
- In $\text{idf}(t)$ the loglog "dampens" the impact of the ratio $\frac{\#\text{documents}}{\#\text{documents with } t}$.
- If a word is very common, the ratio will be close to 1. The log of the ratio will be close to 0.

### Term frequency-inverse document frequency

The **term frequency-inverse document frequency (TF-IDF)** of word $t$ in document $d$ is the product:

$$\begin{aligned}\text{tfidf}(t, d) &= \text{tf}(t, d) \cdot \text{idf}(t) \\ &= \frac{\text{number of occurrences of } t \text{ in } d}{\text{total number of words in } d} \cdot \log\left(\frac{\text{total number of documents}}{\text{number of documents in which } t \text{ appears}}\right)\end{aligned}$$

- If $\text{tfidf}(t, d)$ is large, then $t$ is a good summary of $d$.
  - But to know if $\text{tfidf}(t, d)$ is large, we need to compare it to $\text{tfidf}(t_i, d)$, for several different words $t_i$.
- TF-IDF is a **heuristic** – it has no probabilistic justification.

```
# For a certain word in a sentence
tf = sentences.iloc[1].count('word') / len(sentences.iloc[1].split())
idf = np.log(len(sentences) / sentences.str.contains('word').sum())
tfidf = tf * idf
# TF-IDF for all words in all documents
unique_words = np.unique(sentences.str.split().sum())
tfidf_dict = {}
for word in unique_words:
    re_pat = fr'\b{word}\b'
    tf = sentences.str.count(re_pat) / sentences.str.split().str.len()
    idf = np.log(len(sentences) / sentences.str.contains(re_pat).sum())
    tfidf_dict[word] = tf * idf
# return a DataFrame demonstrating the TF-IDF for all words in all sentences
tfidf = pd.DataFrame(tfidf_dict).set_index(sentences)
```

For a given document, the word with the highest TF-IDF best summarizes that document.

By using `idxmax`, we can find the word with the highest TF-IDF in each sentence.

## Feature Engineering

- A **feature** is a measurable property or characteristic of a phenomenon being observed. ("(explanatory) variable" and "attribute")
- In DataFrames, features typically correspond to **columns**, while rows typically correspond to different individuals.
- There are two types of features: come as part of a dataset v.s we **create**.
- **Feature engineering** is the act of finding **transformations** that transform data into effective **quantitative variables**.
- A feature function $\phi$ (phi, pronounced "fea") is a mapping from raw data to $d$-dimensional space, i.e. $\phi : \text{raw data} \rightarrow \mathbb{R}^d$.
  - If two observations $x_i$ and $x_j$ are "similar" in the raw data space, then $\phi(x_i)$ and $\phi(x_j)$ should also be "similar."

$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - H(x_i)\right)^2$ v.s $\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i - H(x_i)\right)^2}$

```
np.mean((actual - pred) ** 2) # MSE
np.sqrt(np.mean((actual - pred) ** 2)) # RMSE
```

**Key idea: The lower the MSE is, the "better" the model fits the *training* data**.

**Important:** The line that minimizes MSE is the same line that minimizes RMSE and SSE (sum of squared errors).

**Dropping features**

1. When the features **do not contain information** associated with the prediction task.
2. When the feature is **not available at prediction time.**

**Fitting a linear model**

```
from scipy.stats import linregress
lm = linregress(x=galton['father'], y=galton['childHeight'])
# output: LinregressResult(slope=, intercept=, rvalue=, pvalue=, stderr=, intercept_stderr=)
lm.intercept, lm.slope # Use this to predict by calculation
```

**Periodic data**

Transform one column or variable so that the relation between two variables are roughly linear.

# Transform in sklearn

### `Binarizer`

```
sklearn.preprocessing #feature creation
# input: multi-dimensional numpy array (can be df); output: numpy array
from sklearn.preprocessing import Binarizer
binar = Binarizer(threshold=20) # set x=1 if x > thresh, else 0
feat = binar.transform(data) # Binarize all columns in data
```

### `StdScaler`: $z_i = \frac{x_i - \bar{x}}{\sigma_x}$

```
from sklearn.preprocessing import StandardScaler
stdscaler = StandardScaler() # z-scale the data (no parameters)
stdscaler.fit(data) # compute the mean and SD of data # first call the fit method on stdscaler
feat = stdscaler.transform(newdata) # z-scale newdata with mean and SD of data
stdscaler.mean_, stdscaler.var_ # mean and var for each columns
```

### `OneHotEncoder`

So that we don't have to deal with lists within Series, we can **flatten** lists of tags so that there is one column per tag

This process – of converting categorical variables into columns of 1s and 0s – is called **one-hot encoding**.

```
# a function that takes in the list of tags (taglist) for a given quote and returns the one-hot-encoded sequence of 1s and 0s for that quote.
def flatten_tags(taglist):
    return pd.Series({k:1 for k in taglist}, dtype=float)
tags = df['tags'].apply(flatten_tags).fillna(0).astype(int)
tags.head()
```

```
# sklearn also has the function for doing one-hot encoding
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder()
ohe.fit(data)
ohe_features = ohe.transform(data)
ohe.categories_ # unique values (i.e. categories) in each column
ohe_features.toarray() # the resulting matrix is sparse – most of its elements are 0
ohe.get_feature_names() # x0, x1, x2, and x3 correspond to column names in data
ohe.inverse_transform(ohe_features[:10]) # takes a one-hot-encoded matrix and returns a categorical matrix
```

### `QuantileTransformer`

```
from sklearn.preprocessing import QuantileTransformer
qt = QuantileTransformer(n_quantiles=100)
qt.fit(df) qt.transform(df)
```

### `FunctionTransformer`

```
from sklearn.preprocessing import FunctionTransformer
def functionI(parameter): #define function
ft = FunctionTransformer(func=function) # or can put lambda inside
ft.transform(df)
```

# Models in sklearn

### `LinearRegression` : minimizes mean squared error by default.

```
sklearn.linear_models #model creation
from sklearn.linear_model import LinearRegression
lr = LinearRegression() # Create (empty) linear regression model
lr.fit(X, y) # Determines regression coefficients
# ⚠️X needs to be a df to be multi-dimensional (or reshape series/array)⚠️
lr.predict(new_data) # make predictions # Can be 2D with multiple columns
lr.intercept_, lr.coef_ # intercept and coefficient of this linear prediction model
lr.score(data, responses) # Calculate the  R^2 of the LR model
```

**Note**: Once `fit`, estimators like `LinearRegression` are just transformers (`predict` <-> `transform`).

$R^2$ **coefficient of determination, is a measure of the quality of a linear fit.**

- There are a few equivalent ways of computing it, assuming your model has an intercept term:

$R^2 = \frac{\text{var(predicted } y \text{ values)}}{\text{var(actual } y \text{ values)}}$ = `np.var(pred) / np.var(actual)`

$R^2 = [\text{correlation(predicted } y \text{ values, actual } y \text{ values)}]^2$ = non-diagonal entry in `(np.corrcoef(pred, actual)) ** 2`

`lr.score`

- In the simple linear regression case, it is the square of the correlation coefficient, $r$.
- **Key idea:** $R^2$ ranges from 0 to 1. **The closer it is to 1, the better the linear fit is.**
- Interpretation: $R^2$ is the **proportion of variance in** $y$ **that the linear model explains**.

We like linear models with **low RMSE** and **high** $R^2$!

# Pipeline in `sklearn`

- A `Pipeline` object is instantiated using a **list** containing transformer(s) and a model (estimator).

```
pl = Pipeline([feat_trans1, feat_trans2, ..., mdl])
```

- Once a `Pipeline` is instantiated, you can fit **all** steps (transformers and model) using `fit`. `pl.fit(data, responses)`
- To make predictions using **raw (untransformed) data**, use `pl.predict`.

**Creating a `Pipeline`**

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
preproc = ColumnTransformer( # list of tuple
    transformers = [ # first is name, second is transformer, third is list of columns
        ('quant', StandardScaler(), ['total_bill', 'size']),
        ('cat', OneHotEncoder(), ['sex', 'smoker', 'day', 'time'])
    ]#, remainder='passthrough'
)
pl = Pipeline([ # a list of tuples where first is name and second is transformer
    ('preprocessor', preproc),
    ('lin-reg', LinearRegression())
])
pl.fit(tips_features, tips['tip']) # Must fit before predict
pl.predict(tips_features.head())
pl.score(tips_features, tips['tip'])
pl.named_steps['preprocessor'].transform(tips_features) # can access the individual "steps" in pl using the named_steps attribute
```

**Note:** `ColumnTransformer` has a `remainder` argument that you can use to specify what to do with columns that aren't being transformed (`'drop'` or `'passthrough'`).

## Avoiding overfitting

- **Split** our sample into a **training set** and **test set**. + Use **only** the training set to fit the model (i.e. find $w^*$).
- Use the **test** set to evaluate the model's error (RMSE, $R^2$).

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2) # Default 0.25
# Calculating RMSE
from sklearn.metrics import mean_squared_error # built-in RMSE/MSE function
lr = LinearRegression()      /      lr.fit(X_train, y_train)
pred_train = lr.predict(X_train)
mean_squared_error(y_train, pred_train, squared=False) # Root mean square error; True for square mean
```

Since `rmse_train` and `rmse_test` are similar, -> **model is not overfitting** to the training data. Otherwise **not generalize well**.

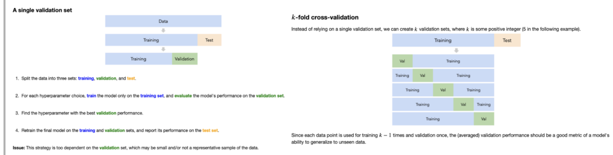
## Bias and variance

- **Bias**: **The expected deviation between a predicted value and an actual value**. Low bias is good
- **Model variance ("variance")**: **The variance of a model's predictions**. Low model variance is good
- Models that have high **bias** are too simple to represent complex relationships in data, and **underfit**.
- Models that have high **variance** are overly complex for the relationships in the data, and vary a lot when fit on different datasets. Such models **overfit** to the training data.

## Parameters vs. hyperparameters

- A **parameter** defines the relationship between variables in a model. **We learn parameters from data**.
- A **hyperparameter** is a parameter that we get to choose **before our model is fit to the data**.

## Cross-validation




```
from sklearn.model_selection import KFold
kfold = KFold(5, shuffle=True, random_state=1)
errs_df = pd.DataFrame()
for train, val in kfold.split(data):
    print(f'train: {data[train]}, validation: {data[val]}')
from sklearn.model_selection import cross_val_score
cross_val_score(estimator, data, target, cv)
#estimator: pipeline(has not alreday been fit); data: training; target: y; cv: k (fold)
```

need to **shuffle** the data first before use cross fold

# Decision Trees

```
from sklearn.tree import DecisionTreeClassifier
dt = DecisionTreeClassifier(max_depth=2) # by default, without restriction, decision trees -> very deep
dt.fit(X_train, y_train)
dt.tree_.max_depth
dt.score(X_train, y_train) or (dt.predict(X_train) == y_train).mean() # Accuracy
```

`.score(X, y)` is $R^2$ in regression; `.score(X, y)` is training accuracy in classification

Decision trees have a tendency to overfit. Make the decision tree "less complex" by limiting the maximum depth.

- If you want to increase the test accuracy, **Reduce** the number of features and **Decrease** the max depth parameter of the decision tree

# Grid search

`GridSearchCV` takes in: an **un-`fit`** instance of an estimator, and and a **dictionary** of hyperparameter values to try,

and performs $k$k-fold cross-validation to find the **combination of hyperparameters with the best average validation performance**. (try all **unique combinations of hyperparameters** )

```
from sklearn.model_selection import GridSearchCV
hyperparameters = {
    'max_depth': [2, 3, 4, 5, 7, 10, 13, 15, 18, None],
    'min_samples_split': [2, 3, 5, 7, 10, 15, 20],
    'criterion': ['gini', 'entropy']
} # there are 140 combinations of hyperparameters (len(max) * len(min) * len(cri))
searcher = GridSearchCV(DecisionTreeClassifier(), hyperparameters, cv=5)
searcher.fit(X_train, y_train)
searcher.best_params_ # best combination of hyperparameters to use
searcher.cv_results_['mean_test_score'] # array of length 140
# All of the intermediate results – validation accuracies for each fold, mean validation accuaries
searcher.predict(X_train)
searcher.score(X_test, y_test)
```

## Multicollinearity

Redundant features: Data in different units, will not change the RMSE if we use data in other unit as one more feature

In other words, multicollinearity occurs when **a feature can be predicted using a linear combination of other features, fairly accurately**.

**Multicollinearity doesn't impact a model's predictions!**

Manually remove highly correlated features. or Use a dimensionality reduction technique (such as PCA) to reduce dimensions.

**Multicollinearity is present when performing one-hot encoding**

```
pd.get_dummies(tips_features, drop_first=True) # drop one column per categorical feature.
```

# Modeling using text features

### `CountVectorizer`

```
from sklearn.feature_extraction.text import CountVectorizer
example_corp = ['hey hey hey my name is billy',
                'hey billy how is your dog billy']
count_vec = CountVectorizer()
count_vec.fit(example_corp)
count_vec.vocabulary_ # learned a vocabulary from the corpus we fit it on
count_vec.transform(example_corp).toarray()
```

### `RandomForestClassifier`

- A "random forest" is a combination (or **ensemble**) of decision trees, each fit on a different **bootstrapped** resample of the training data.

```
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
X_train, X_test, y_train, y_test = train_test_split(X, y)
pl = Pipeline([
    ('cv', CountVectorizer()),
    ('clf', RandomForestClassifier(max_depth=8, n_estimators=7)) # Uses 7 separate decision trees
])
pl.fit(X_train, y_train)
pl.score(X_train, y_train) / pl.score(X_test, y_test)
```

# Classifier evaluation

## Outcomes in binary classification

When performing **binary** classification, there are four possible outcomes.

(Note: A "positive prediction" is a prediction of 1, and a "negative prediction" is a prediction of 0.)

| Outcome of Prediction | Definition | True Class |
|---|---|---|
| **True** positive (TP) ✅ | The predictor **correctly** predicts the positive class. | P |
| False negative (FN) ❌ | The predictor incorrectly predicts the negative class. | P |
| **True** negative (TN) ✅ | The predictor **correctly** predicts the negative class. | N |
| False positive (FP) ❌ | The predictor incorrectly predicts the positive class. | N |

⬇️

| | Predicted Negative | Predicted Positive |
|---|---|---|
| **Actually Negative** | TN ✅ | FP ❌ |
| **Actually Positive** | FN ❌ | TP ✅ |

*The **confusion matrix** above is organized the same way that `sklearn`'s confusion matrices are (but differently than in the wolf example).*

Note that in the four acronyms – TP, FN, TN, FP – the **first letter** is whether the prediction is correct, and the **second letter** is what the prediction is.

$\text{accuracy} = \frac{TP+TN}{TP+FP+FN+TN}$

$\text{recall} = \frac{TP}{TP+FN}$: **recall** of a binary classifier is the proportion of **actually positive instances** that are correctly classified.

$\text{precision} = \frac{TP}{TP+FP}$: The **precision** of a binary classifier is the proportion of **predicted positive instances** that are correctly classified. We'd like this number to be as close to 1 (100%) as possible.

- If simply predict all as one result, TP decrease TN will increase, or TP increase TN will decrease.

**Precision and recall:** $\text{precision} = \frac{TP}{TP+FP} \qquad \text{recall} = \frac{TP}{TP+FN}$

🤔 **Question:** When might high **precision** be more important than high recall?

🙋 **Answer:** For instance, in deciding whether or not someone committed a crime. Here, **false positives are really bad** – they mean that an innocent person is charged! **More false negative than false positive**

🤔 **Question:** When might high **recall** be more important than high precision?

🙋 **Answer:** For instance, in medical tests. Here, **false negatives are really bad** – they mean that someone's disease goes undetected! **More false positive than false negative**