Northeastern University

DS 5110 – Lecture 4
Pandas

Roi Yehoshua

Agenda

Northeastern University

- Pandas objects: Series, DataFrame
- Selecting data
- Statistical methods
- Loading data from CSV and JSON files
- Grouping
- Merging data frames
- Plotting data frames

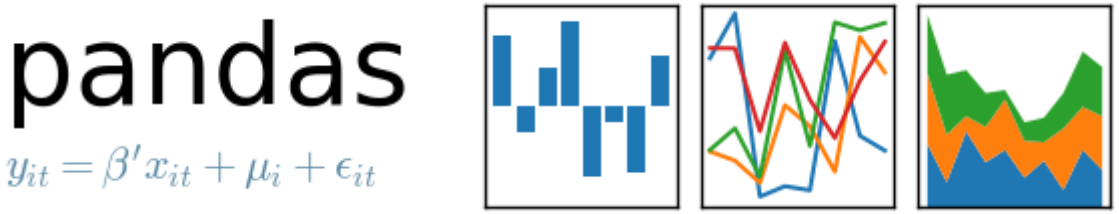
Pandas

Northeastern University

- Pandas is a high-performance library for data analysis in Python
- Provides methods to handle data sets in different formats: tabular, CSV, JSON, time series, databases (SQL), and others
- Built on top of the NumPy library, thus delivers fast performance

- Typically the pandas library is imported under the alias pd:

```
import pandas as pd
```



Pandas Data Objects

Northeastern University

- The primary data objects in pandas are:
  - Series – a one-dimensional array of indexed data
  - DataFrame – a multi-dimensional table, made up of collection of series

Series

Northeastern University

- A Series is a one-dimensional array of indexed data (of any NumPy data type)
- The syntax for creating a Series data structure is:

```
pd.Series(iterable, index=idx)
```

  - If an index is not specified, a default index [0,... n-1], is created

```
ser1 = pd.Series([15, 8, 33])
ser1
```

```
0    15
1     8
2    33
dtype: int64
```

```
ser1[1]
```

```
8
```

Series

Northeastern University

- Example for creating a Series with index:

```
ser2 = pd.Series([15, 8, 33],
                 index=['a', 'b', 'c'])
ser2
```

```
a    15
b     8
c    33
dtype: int64
```

- We can access each row in the Series using its index or its row number:

```
ser2[1]
```

```
8
```

```
ser2['b']
```

```
8
```

Series

Northeastern University

- A Series object can be constructed directly from a Python dictionary:

```
population_data = {
    'California': 39538223,
    'Texas': 29145505,
    'Florida': 21538187,
    'New York': 20201249,
    'Massachusetts': 7029917
}

population = pd.Series(population_data)
population
```

```
California       39538223
Texas            29145505
Florida          21538187
New York         20201249
Massachusetts     7029917
dtype: int64
```

Slicing

Northeastern University

- Slicing can be used both on the indexes and the rows locations

```
population[1:4]
```

```
Texas       29145505
Florida     21538187
New York    20201249
dtype: int64
```

```
population['Texas':'New York']
```

```
Texas       29145505
Florida     21538187
New York    20201249
dtype: int64
```

- Note that when slicing with labels the endpoint is inclusive

Operations on Series

Northeastern University

- Since Series is based on NumPy arrays, it supports many of its operations:

```
ser = pd.Series([15, 8, 33])

ser * 2
```

```
0    30
1    16
2    66
dtype: int64
```

```
ser[1:3]
```

```
1     8
2    33
dtype: int64
```

```
import numpy as np

np.mean(ser), np.std(ser)
```

```
(18.666666666666668, 10.530379332620875)
```

Index Alignment

Northeastern University

- When performing operations between two Series, Pandas aligns their indices

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967})
population = pd.Series({'California': 39776830, 'Texas': 28704330,
                        'Florida': 21312211})
```

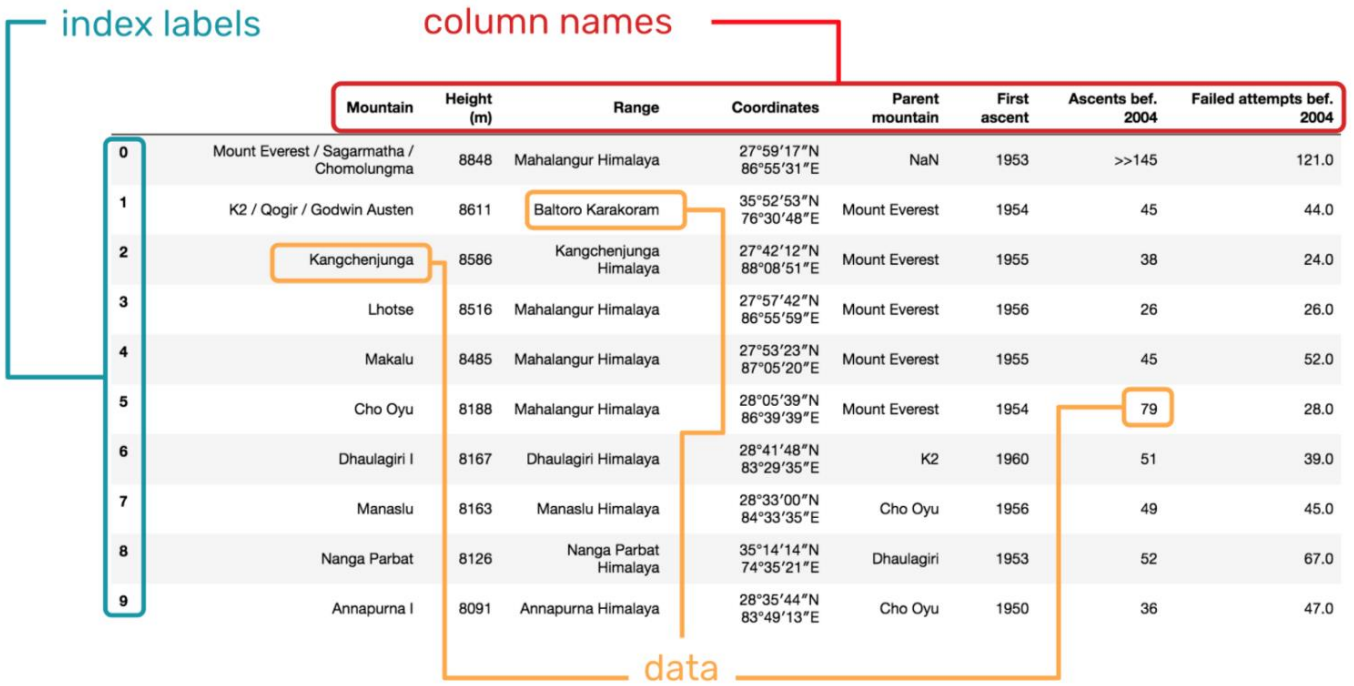
```
population / area
```

```
Alaska              NaN
California    93.820580
Florida             NaN
Texas         41.261892
dtype: float64
```

- The resulting Series contains the union of indices of the two Series
- Any item for which one or the other doesn't have an entry is marked with NaN

DataFrame

- DataFrame is the most commonly used data structure in pandas
- It is an analog of a 2D array with flexible row indices and column names
- Each DataFrame column is a Series structure



	Mountain	Height (m)	Range	Coordinates	Parent mountain	First ascent	Ascents bef. 2004	Failed attempts bef. 2004
0	Mount Everest / Sagarmatha / Chomolungma	8848	Mahalangur Himalaya	27°59′17″N 86°55′31″E	NaN	1953	>>145	121.0
1	K2 / Qogir / Godwin Austen	8611	Baltoro Karakoram	35°52′53″N 76°30′48″E	Mount Everest	1954	45	44.0
2	Kangchenjunga	8586	Kangchenjunga Himalaya	27°42′12″N 88°08′51″E	Mount Everest	1955	38	24.0
3	Lhotse	8516	Mahalangur Himalaya	27°57′42″N 86°55′59″E	Mount Everest	1956	26	26.0
4	Makalu	8485	Mahalangur Himalaya	27°53′23″N 87°05′20″E	Mount Everest	1955	45	52.0
5	Cho Oyu	8188	Mahalangur Himalaya	28°05′39″N 86°39′39″E	Mount Everest	1954	79	28.0
6	Dhaulagiri I	8167	Dhaulagiri Himalaya	28°41′48″N 83°29′35″E	K2	1960	51	39.0
7	Manaslu	8163	Manaslu Himalaya	28°33′00″N 84°33′35″E	Cho Oyu	1956	49	45.0
8	Nanga Parbat	8126	Nanga Parbat Himalaya	35°14′14″N 74°35′21″E	Dhaulagiri	1953	52	67.0
9	Annapurna I	8091	Annapurna Himalaya	28°35′44″N 83°49′13″E	Cho Oyu	1950	36	47.0

DataFrame Construction Methods

Northeastern University

- The following table shows the possible data inputs to DataFrame constructor:

Type	Description
dict of Series	Each Series becomes a column. Indexes from each Series are unioned together to form the result's row index if no explicit index is passed.
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the "dict of Series" case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame's column labels.
2D ndarray	A matrix of data, passing optional row and column labels
NumPy structured array	Treated as the "dict of arrays" case
list of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed

DataFrame from a Dictionary of Lists

Northeastern University

- The easiest way to create a DataFrame is from a dictionary of equal-length lists
- Each list defines one of the columns in the DataFrame
- The dictionary keys will become the column names

```
states_data = {
    'name': ['California', 'Massachusetts', 'Texas', 'New York', 'Florida'],
    'capital': ['Sacramento', 'Boston', 'Austin', 'Albany', 'Tallahassee'],
    'population': [39512223, 6892503, 28995881, 19453561, 21477737],
    'area': [163695, 10554, 268596, 54555, 65758]
}

states = pd.DataFrame(states_data)
states
```

	name	capital	population	area
0	California	Sacramento	39512223	163695
1	Massachusetts	Boston	6892503	10554
2	Texas	Austin	28995881	268596
3	New York	Albany	19453561	54555
4	Florida	Tallahassee	21477737	65758

DataFrame from a Dictionary of Lists

Northeastern University

- As in Series, you can specify an index that defines the row labels:

```
states_data = {
    'capital': ['Sacramento', 'Boston', 'Austin', 'Albany', 'Tallahassee'],
    'population': [39512223, 6892503, 28995881, 19453561, 21477737],
    'area': [163695, 10554, 268596, 54555, 65758]
}
names = ['California', 'Massachusetts', 'Texas', 'New York', 'Florida']

states = pd.DataFrame(states_data, index=names)
states
```

	capital	population	area
California	Sacramento	39512223	163695
Massachusetts	Boston	6892503	10554
Texas	Austin	28995881	268596
New York	Albany	19453561	54555
Florida	Tallahassee	21477737	65758

DataFrame from a 2D NumPy array

Northeastern University

- Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['Column A', 'Column B'],
             index=['a', 'b', 'c'])
```

	Column A	Column B
a	0.188171	0.090570
b	0.308763	0.254368
c	0.630323	0.761876

- If omitted, an integer index will be used for each:

```
pd.DataFrame(np.random.rand(3, 3))
```

	0	1	2
0	0.237744	0.876218	0.583536
1	0.059770	0.177468	0.564984
2	0.510760	0.789046	0.486915

DataFrame Attributes

Northeastern University

- DataFrame has number of attributes enabling you to access its metadata:
  - **index** gives the row labels
  - **columns** give the column labels
  - **dtypes** gives the data types of the columns
  - **shape** gives the axis dimensions as in NumPy
  - **values** gives the data in the DataFrame as a 2D NumPy array

DataFrame Attributes

Northeastern University

```
states.index
```

```
Index(['California', 'Massachusetts', 'Texas', 'New York', 'Florida'], dtype='object')
```

```
states.columns
```

```
Index(['capital', 'population', 'area'], dtype='object')
```

```
states.dtypes
```

```
capital       object
population     int64
area           int64
dtype: object
```

```
states.shape
```

```
(5, 3)
```

```
states.values
```

```
array([['Sacramento', 39512223, 163695],
       ['Boston', 6892503, 10554],
       ['Austin', 28995881, 268596],
       ['Albany', 19453561, 54555],
       ['Tallahassee', 21477737, 65758]], dtype=object)
```

Data Selection

Northeastern University

- Pandas supports the standard NumPy methods for accessing data in arrays (indexing, slicing, masking, etc.)
- In addition, it provides additional data access methods (.loc and .iloc) that are more optimized for data frames and provide additional options
  - It is recommended to use them for production code

Selection by Label

Northeastern University

- **df.loc[]** selects a group of rows and columns using the index and column names
- df.loc[] may accept two arguments for the rows and the columns indices, or only one argument for the row index
- In both cases, the allowed inputs are:
  - A single label, e.g., ‘a’
  - A list of labels, e.g., [‘a’, ‘b’]
  - A slice object with labels, e.g., ‘a’:’f’
  - A Boolean mask, e.g., ‘a’ > 5

Selection by Label

Northeastern University

```
states.loc['Texas']
```

```
capital         Austin
population    28995881
area            268596
Name: Texas, dtype: object
```

```
states.loc['Texas', 'area']
```

```
268596
```

```
states.loc['California':'Texas', 'area']
```

```
California       163695
Massachusetts     10554
Texas            268596
Name: area, dtype: int64
```

with label slicing, both endpoints are included

```
states.loc['Texas', ['area', 'population']]
```

```
area            268596
population    28995881
Name: Texas, dtype: object
```

```
states.loc[:, ['area', 'population']]
```

	area	population
California	163695	39512223
Massachusetts	10554	6892503
Texas	268596	28995881
New York	54555	19453561
Florida	65758	21477737

```
states.loc[states['area'] > 100000, 'area']
```

```
California    163695
Texas         268596
Name: area, dtype: int64
```

Selection by Position

Northeastern University

- **df.iloc[]** selects a group of rows and columns based on their integer positions
- df.iloc[] may accept two arguments for the rows and the columns indices, or only one argument for the row index
- In both cases, the allowed inputs are:
  - An integer, e.g., 5
  - A list of integers, e.g., [4, 3, 0]
  - A slice object with ints, e.g., 1:7

Selection by Position

Northeastern University

```
states.iloc[1]
```

```
capital        Boston
population    6892503
area            10554
Name: Massachusetts, dtype: object
```

```
states.iloc[1, 1]
```

```
6892503
```

```
states.iloc[1:4, 0:2]
```

	capital	population
Massachusetts	Boston	6892503
Texas	Austin	28995881
New York	Albany	19453561

```
states.iloc[[1, 2, 4], [0, 2]]
```

	capital	area
Massachusetts	Boston	10554
Texas	Austin	268596
Florida	Tallahassee	65758

```
states.iloc[1:4, [0, 2]]
```

	capital	area
Massachusetts	Boston	10554
Texas	Austin	268596
New York	Albany	54555

Changing Values

Northeastern University

- You can use .loc and .iloc to assign values to specific cells:

```
states.loc['Florida', 'population'] += 1000
```

```
states
```

	capital	population	area
California	Sacramento	39512223	163695
Massachusetts	Boston	6892503	10554
Texas	Austin	28995881	268596
New York	Albany	19453561	54555
Florida	Tallahassee	21478737	65758

Selecting Columns

Northeastern University

- To select a single column you can also use df[column_name] or df.column_name:

```
states['population']
```

```
California       39512223
Massachusetts     6892503
Texas            28995881
New York         19453561
Florida          21477737
Name: population, dtype: int64
```

```
states.area
```

```
California       163695
Massachusetts     10554
Texas            268596
New York          54555
Florida           65758
Name: area, dtype: int64
```

Class Exercise

Northeastern University

- Create the following DataFrame:

	animal	size	weight	adult
0	cat	S	8	False
1	dog	S	10	False
2	cat	M	11	False
3	fish	M	1	False
4	dog	M	20	False
5	cat	L	12	True
6	cat	L	12	True

- Select the first 3 rows of the DataFrame
- Select the columns ‘animal’ and ‘weight’
- Select the weights of the animals in rows 3 and 5
- Select all the cats with large size
- Change the weight of the last animal to 15

Adding Columns

Northeastern University

- You can add a column to the DataFrame by assigning a list or a NumPy array to a new column name:

```
states['abbr'] = ['CA', 'MA', 'TX', 'NY', 'FL']
```

```
states
```

	capital	population	area	abbr
California	Sacramento	39512223	163695	CA
Massachusetts	Boston	6892503	10554	MA
Texas	Austin	28995881	268596	TX
New York	Albany	19453561	54555	NY
Florida	Tallahassee	21478737	65758	FL

Adding Columns

Northeastern University

- The new column can be based on existing columns of the DataFrame:

```
states['density'] = states['population'] / states['area']
```

```
states
```

	capital	population	area	abbr	density
California	Sacramento	39512223	163695	CA	241.377092
Massachusetts	Boston	6892503	10554	MA	653.070210
Texas	Austin	28995881	268596	TX	107.953510
New York	Albany	19453561	54555	NY	356.586216
Florida	Tallahassee	21478737	65758	FL	326.633064

Deleting Rows and Columns

Northeastern University

- **drop**(*labels*, *axis=0*) allows you to drop specified labels from the rows or columns
  - Use axis=0 to drop rows and axis=1 to drop columns
  - The method returns a copy of the DataFrame, unless inplace=True is specified

```
states.drop(['California', 'Texas'])
```

	capital	population	area	abbr	density
Massachusetts	Boston	6892503	10554	MA	653.070210
New York	Albany	19453561	54555	NY	356.586216
Florida	Tallahassee	21478737	65758	FL	326.633064

```
states.drop('capital', axis=1)
```

	population	area	abbr	density
California	39512223	163695	CA	241.377092
Massachusetts	6892503	10554	MA	653.070210
Texas	28995881	268596	TX	107.953510
New York	19453561	54555	NY	356.586216
Florida	21478737	65758	FL	326.633064

Index Setting and Resetting

Northeastern University

- **reset_index()** resets the index of the DataFrame and uses the default one instead
- For example, we can move the index of the population DataFrame to become a new column

```
states.reset_index()
```

	index	capital	population	area
0	California	Sacramento	39512223	163695
1	Massachusetts	Boston	6892503	10554
2	Texas	Austin	28995881	268596
3	New York	Albany	19453561	54555
4	Florida	Tallahassee	21477737	65758

Index Setting and Resetting

Northeastern University

- **set_index()** is the opposite operation which sets a new index using one of the columns

```
states.set_index('capital')
```

capital	population	area
Sacramento	39512223	163695
Boston	6892503	10554
Austin	28995881	268596
Albany	19453561	54555
Tallahassee	21477737	65758

Sorting

Northeastern University

- **sort_index**(*axis=0, ascending=True*) sorts the DataFrame by its row or column index
  - Use axis=0 to sort by the row index and axis=1 to sort by the column names
  - The *ascending* argument can be used to change the sorting order

```
states.sort_index()
```

	capital	population	area
California	Sacramento	39512223	163695
Florida	Tallahassee	21477737	65758
Massachusetts	Boston	6892503	10554
New York	Albany	19453561	54555
Texas	Austin	28995881	268596

```
states.sort_index(axis=1)
```

	area	capital	population
California	163695	Sacramento	39512223
Massachusetts	10554	Boston	6892503
Texas	268596	Austin	28995881
New York	54555	Albany	19453561
Florida	65758	Tallahassee	21477737

Sorting

Northeastern University

- **sort_values**(*by*, *axis=0*, *ascending=True*) sorts by the values along the specified axis
  - *by* is a label or list of labels to sort by

```
states.sort_values('area')
```

	capital	population	area
Massachusetts	Boston	6892503	10554
New York	Albany	19453561	54555
Florida	Tallahassee	21477737	65758
California	Sacramento	39512223	163695
Texas	Austin	28995881	268596

Sorting

Northeastern University

- **nlargest**(*n*, *columns*) returns the *n* rows with the largest values in the specified columns

```
states.nlargest(3, 'population')
```

	capital	population	area
California	Sacramento	39512223	163695
Texas	Austin	28995881	268596
Florida	Tallahassee	21477737	65758

Handling Missing Data

Northeastern University

- Pandas considers np.nan and Python's None object as missing values
- Referred to as NA (Not Available) values
- By default they are not included in computations

```
states.loc[['Florida', 'Texas'], 'area'] = np.nan
```

```
states
```

	capital	population	area
California	Sacramento	39512223	163695.0
Massachusetts	Boston	6892503	10554.0
Texas	Austin	28995881	NaN
New York	Albany	19453561	54555.0
Florida	Tallahassee	21477737	NaN

```
states['area'].mean()
```

```
76268.0
```

Handling Missing Data

Northeastern University

- There are several useful methods for detecting, removing, and replacing null values

Method	Description
isna(), isnull()	Generate a boolean mask indicating missing values
dropna()	Return a filtered version of the data
fillna()	Return a copy of the data with missing values filled or imputed

Handling Missing Data

Northeastern University

- The method **isna()** can be used to detect NA values

```
states.isna()
```

	capital	population	area
California	False	False	False
Massachusetts	False	False	False
Texas	False	False	True
New York	False	False	False
Florida	False	False	True

Handling Missing Data

Northeastern University

- Use **dropna**(*axis=0, how='any'*) to drop any rows that have missing data

```
states.dropna()
```

	capital	population	area
California	Sacramento	39512223	163695.0
Massachusetts	Boston	6892503	10554.0
New York	Albany	19453561	54555.0

- The *how* argument determines if a row or column is removed from the DataFrame, when it has at least one NA or all NA:
  - 'any': If any NA values are present, drop that row or column
  - 'all': If all values are NA, drop that row or column

Handling Missing Data

Northeastern University

- You can use the method **fillna()** to fill NA entries with some value

```
states.fillna(0)
```

	capital	population	area
California	Sacramento	39512223	163695.0
Massachusetts	Boston	6892503	10554.0
Texas	Austin	28995881	0.0
New York	Albany	19453561	54555.0
Florida	Tallahassee	21477737	0.0

Statistical Methods

Northeastern University

- Pandas provides a set of common statistical methods

Method	Description
count()	Number of non-NA values
min(), max()	Minimum and maximum values
argmin(), argmax()	The integer positions of the minimum and maximum values
idxmin(), idxmax()	The indices of the minimum and maximum values
sum()	Sum
cumsum()	Cumulative sum
mean()	Mean of values
median()	Arithmetic median (50% quantile)
var()	Sample variance
std()	Sample standard deviation
cov()	Sample covariance
corr()	Sample corrleation

Statistical Methods

Northeastern University

- For example, assume we have the following DataFrame with random numbers:

```
df = pd.DataFrame(np.random.rand(6, 4), columns=list('ABCD'))
df
```

	A	B	C	D
0	0.118274	0.639921	0.143353	0.944669
1	0.521848	0.414662	0.264556	0.774234
2	0.456150	0.568434	0.018790	0.617635
3	0.612096	0.616934	0.943748	0.681820
4	0.359508	0.437032	0.697631	0.060225
5	0.666767	0.670638	0.210383	0.128926

Statistical Methods

Northeastern University

- **df.mean()** returns the mean of each column:

```
df.mean()
```

```
A    0.455774
B    0.557937
C    0.379743
D    0.534585
dtype: float64
```

- To apply the same operation on the rows, specify axis=1:

```
df.mean(axis=1)
```

```
0    0.461554
1    0.493825
2    0.415252
3    0.713650
4    0.388599
5    0.419178
dtype: float64
```

Statistical Methods

Northeastern University

- **df.idxmax()** returns the index of the row with the highest value

```
df.idxmax()
```

```
A    5
B    5
C    3
D    0
dtype: int64
```

Covariance

Northeastern University

- The covariance of two attributes measures the degree to which the two attributes vary together
- The **covariance matrix** *S* defines the covariances between each pair of attributes
- s_{ij} is the covariance of the i^{th} and j^{th} attributes:

$$s_{ij} = \text{cov}(x_i, x_j) = \frac{1}{m-1} \sum_{k=1}^{m} (x_{ki} - \overline{x_i})(x_{kj} - \overline{x_j})$$

  - x_{ki} and x_{kj} are the values of the i^{th} and j^{th} attributes for the k^{th} object
- A value near 0 means that there is no (linear) relationship between the attributes

Correlation

Northeastern University

- The covariance of two attributes depend on the magnitudes of the variables
  - Thus, we can't use the covariance values to judge the degree of their relationship
- The **Pearson's correlation coefficient** of two attributes x_i and x_j is their covariance normalized by the product of their standard deviations

$$r_{ij} = \text{correlation}(x_i, x_j) = \frac{\text{cov}(x_i, x_j)}{\sigma_i \sigma_j}$$

- The correlation coefficient lies between -1 and 1
  - $r_{ij} > 0$: x_i and x_j are positively correlated (x_i's values increase as x_j's)
    - The higher, the stronger correlation
  - $r_{ij} = 0$: x_i and x_j are independent
  - $r_{ij} < 0$: x_i and x_j negatively correlated

Correlation and Covariance

Northeastern University

- DataFrame's **corr()** and **cov()** methods return the correlation or covariance matrix as a DataFrame, respectively:

```
states.cov()
```

	population	area
population	1.456488e+14	9.249620e+11
area	9.249620e+11	1.073056e+10

```
states.corr()
```

	population	area
population	1.000000	0.739877
area	0.739877	1.000000

Value Counts (Histogramming)

Northeastern University

- The **value_counts()** Series method computes the value frequencies in the Series

```
data = np.random.randint(0, 5, size=50)
data
```

```
array([3, 2, 3, 1, 1, 0, 1, 4, 2, 4, 4, 2, 4, 2, 0, 1, 0, 4, 2, 2, 1, 0,
       2, 0, 0, 4, 0, 1, 0, 3, 4, 1, 1, 1, 4, 3, 4, 1, 0, 2, 1, 4, 2, 4,
       2, 4, 2, 2, 3, 3])
```

```
s = pd.Series(data)
```

```
s.value_counts()
```

```
4    12
2    12
1    11
0     9
3     6
dtype: int64
```

Discretization

Northeastern University

- Continuous data is often discretized or otherwised separated into “bins” for analysis
- Suppose you have data about a group of people in a study, and you want to group them into discrete age buckets:

```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

- Let’s divide these into bins of 18-25, 26-35, 35-60, and >60
- To do so, you can use the function **pd.cut()**:

```
bins = [18, 25, 35, 60, 100]
cats = pd.cut(ages, bins)
cats
```

```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100],
(35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

Discretization

Northeastern University

- A closely related function, **pd.qcut()**, bins the data based on quantiles
- It creates equal-sized bins

```
data = np.random.randn(1000) # Normally distributes
cats = pd.qcut(data, 4) # Cut into quanrtiles
cats
```

```
[(0.653, 3.28], (0.653, 3.28], (0.653, 3.28], (0.0156, 0.653], (0.0156, 0.653],
..., (-2.991, -0.613], (0.653, 3.28], (0.653, 3.28], (-0.613, 0.0156], (0.653,
3.28]]
Length: 1000
Categories (4, interval[float64]): [(-2.991, -0.613] < (-0.613, 0.0156] < (0.01
56, 0.653] < (0.653, 3.28]]
```

```
pd.value_counts(cats)
```

```
(0.653, 3.28]       250
(0.0156, 0.653]     250
(-0.613, 0.0156]    250
(-2.991, -0.613]    250
dtype: int64
```

Applying Functions

Northeastern University

- **apply**(*func*, *axis=0*) allows you to apply any function along an axis of the DataFrame
  - Use axis 0 to apply the function on the columns and axis 1 to apply the function on the rows

```
df.apply(np.mean)
```

```
A    0.543673
B    0.533104
C    0.489543
D    0.583819
dtype: float64
```

```
df.apply(lambda x: x.max() - x.min())
```

```
A    0.813101
B    0.532671
C    0.894603
D    0.871448
dtype: float64
```

Applying Functions

Northeastern University

- The Series **map**(*func*) method allows you to apply an element-wise function on a specific column

```
df['E'] = df['A'].map(lambda x: 10 * x)
df
```

	A	B	C	D	E
0	0.408302	0.287960	0.441024	0.739285	4.083015
1	0.443191	0.399741	0.395765	0.611734	4.431912
2	0.550124	0.432249	0.130995	0.617387	5.501242
3	0.704006	0.539369	0.911959	0.033687	7.040060
4	0.984759	0.718674	0.976058	0.905135	9.847589
5	0.171657	0.820631	0.081455	0.595684	1.716574

Applying Functions

Northeastern University

- You can also use a dictionary to map the values in the column into new values:

```
s = pd.Series(['cat', 'dog', 'rabbit'])
s.map({'cat': 'kitten', 'dog': 'puppy'})
```

```
0    kitten
1     puppy
2       NaN
dtype: object
```

- Values that are not found in the dict are converted to NaN, unless the dict has a default value (e.g., defaultdict)

Class Exercise

Northeastern University

- In the animals DataFrame:
  - Compute the mean weight of all the dogs
  - Display the details of the heaviest animal
  - Display the number of adult animals
  - Display the number of animals from each type
  - Convert the size column to be numeric (i.e., map the values ‘S’, ‘M’, ‘L’ to 0, 1, 2)
  - Compute the correlation between the size of the animals and their weight

String Methods

Northeastern University

- Pandas provides a comprehensive set of vectorized string operations
- These are accessed via the **str** attribute of the Series object
- They generally have names matching the equivalent Python's built-in string methods:

`len()`	`lower()`	`translate()`	`islower()`
`ljust()`	`upper()`	`startswith()`	`isupper()`
`rjust()`	`find()`	`endswith()`	`isnumeric()`
`center()`	`rfind()`	`isalnum()`	`isdecimal()`
`zfill()`	`index()`	`isalpha()`	`split()`
`strip()`	`rindex()`	`isdigit()`	`rsplit()`
`rstrip()`	`capitalize()`	`isspace()`	`partition()`
`lstrip()`	`swapcase()`	`istitle()`	`rpartition()`

String Methods

Northeastern University

- Example:

```
states['capital'].str.lower()
```

```
California        sacramento
Massachusetts         boston
Texas                 austin
New York              albany
Florida          tallahassee
Name: capital, dtype: object
```

```
states['capital'].str.len()
```

```
California       10
Massachusetts     6
Texas             6
New York          6
Florida          11
Name: capital, dtype: int64
```

String Methods

Northeastern University

- **str.split()** returns a list of values for each string in the Series:

```
actors = pd.Series(['Jack Nicholson', 'Meryl Streep', 'Tom Hanks', 'Kate Winslet'])
actors.str.split()
```

```
0    [Jack, Nicholson]
1      [Meryl, Streep]
2         [Tom, Hanks]
3      [Kate, Winslet]
dtype: object
```

- Passing **expand=True** to split() expands the splitted strings into separate columns:

```
actors.str.split(expand=True)
```

	0	1
0	Jack	Nicholson
1	Meryl	Streep
2	Tom	Hanks
3	Kate	Winslet

String Indexing and Slicing

Northeastern University

- .str() supports Python's normal indexing and slicing syntax

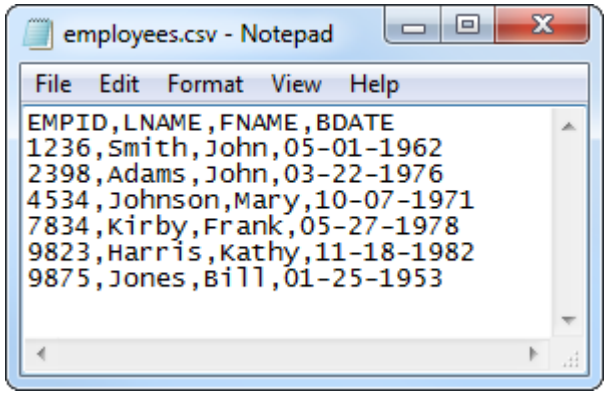
```
actors.str[0:4]
```

```
0    Jack
1    Mery
2    Tom
3    Kate
dtype: object
```

Loading Data from CSV Files

Northeastern University

- CSV (Comma-Separated Values) is a text file that saves data in a tabular format
- CSV files can be used with any spreadsheet program, such as Microsoft Excel or Google Spreadsheets



- You can read data from a CSV into a DataFrame using **pd.read_csv**(*filepath*)

Loading Data from CSV Files

Northeastern University

- For example, let's load the data from the file flights.csv (available at the course website) which contains 58,493 rows with data about flights

```
flights = pd.read_csv('flights.csv')
flights
```

	Month	Day	Weekday	Airline	Origin	Dest	AirTime	Distance	ArrivalDelay	Diverted	Cancelled
0	1	1	4	WN	LAX	SLC	94.0	590	65.0	0	0
1	1	1	4	UA	DEN	IAD	154.0	1452	-13.0	0	0
2	1	1	4	MQ	DFW	VPS	85.0	641	35.0	0	0
3	1	1	4	AA	DFW	DCA	126.0	1192	-7.0	0	0
4	1	1	4	WN	LAX	MCI	166.0	1363	39.0	0	0
...	...	...	...	...	...	...	...	...	...	...	...
58487	12	31	4	AA	SFO	DFW	166.0	1464	-19.0	0	0
58488	12	31	4	F9	LAS	SFO	71.0	414	4.0	0	0
58489	12	31	4	OO	SFO	SBA	46.0	262	-5.0	0	0
58490	12	31	4	WN	MSP	ATL	124.0	907	34.0	0	0
58491	12	31	4	OO	SFO	BOI	73.0	522	-1.0	0	0

58492 rows × 11 columns

Additional Arguments

Northeastern University

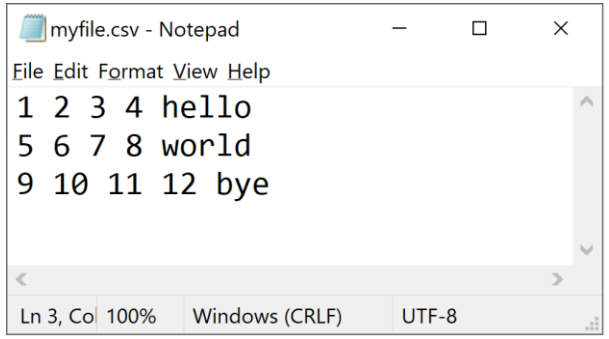
- pd.read_csv() has many additional arguments to help you handle the wide variety of exception file formats that might occur:

Argument	Description
sep	Character sequence or regular expression to use to split fields in each row
header	Row number to use as column names. Defaults to 0 (first row), but should be None if there is no header row.
names	List of column names for result, combine with header=None
index_col	Column numbers or names to use as the row index in the result
skiprows	Number of rows at beginning of file to ignore
skip_footer	Number of lines to ignore at end of file
na_values	Sequence of values to replace with NA
comment	Character or characters to split comments off the end of lines

Additional Arguments

Northeastern University

- For example, the following file doesn't have an header and the separator is space



- To read it, you can allow pandas to assign default column names, or you can specify names yourself:

```
pd.read_csv('myfile.csv', sep=' ', header=None)
```

	0	1	2	3	4
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	bye

```
pd.read_csv('myfile.csv', sep=' ',
            names=['a', 'b', 'c', 'd', 'e'])
```

	a	b	c	d	e
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	bye

Viewing the Data

Northeastern University

- You can view the top and bottom rows of the frame by using **head()** and **tail()**
  - By default they display the first/last 5 rows

```
flights.head()
```

	Month	Day	Weekday	Airline	Origin	Dest	AirTime	Distance	ArrivalDelay	Diverted	Cancelled
0	1	1	4	WN	LAX	SLC	94.0	590	65.0	0	0
1	1	1	4	UA	DEN	IAD	154.0	1452	-13.0	0	0
2	1	1	4	MQ	DFW	VPS	85.0	641	35.0	0	0
3	1	1	4	AA	DFW	DCA	126.0	1192	-7.0	0	0
4	1	1	4	WN	LAX	MCI	166.0	1363	39.0	0	0

```
flights.tail()
```

	Month	Day	Weekday	Airline	Origin	Dest	AirTime	Distance	ArrivalDelay	Diverted	Cancelled
58487	12	31	4	AA	SFO	DFW	166.0	1464	-19.0	0	0
58488	12	31	4	F9	LAS	SFO	71.0	414	4.0	0	0
58489	12	31	4	OO	SFO	SBA	46.0	262	-5.0	0	0
58490	12	31	4	WN	MSP	ATL	124.0	907	34.0	0	0
58491	12	31	4	OO	SFO	BOI	73.0	522	-1.0	0	0

Viewing the Data

Northeastern University

- **info()** prints about the DataFrame including the index dtype and columns, non-null values and memory usage

```
flights.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 58492 entries, 0 to 58491
Data columns (total 11 columns):
Month           58492 non-null int64
Day             58492 non-null int64
Weekday         58492 non-null int64
Airline         58492 non-null object
Origin          58492 non-null object
Dest            58492 non-null object
AirTime         57474 non-null float64
Distance        58492 non-null int64
ArrivalDelay    57474 non-null float64
Diverted        58492 non-null int64
Cancelled       58492 non-null int64
dtypes: float64(2), int64(6), object(3)
memory usage: 4.9+ MB
```

Viewing the Data

Northeastern University

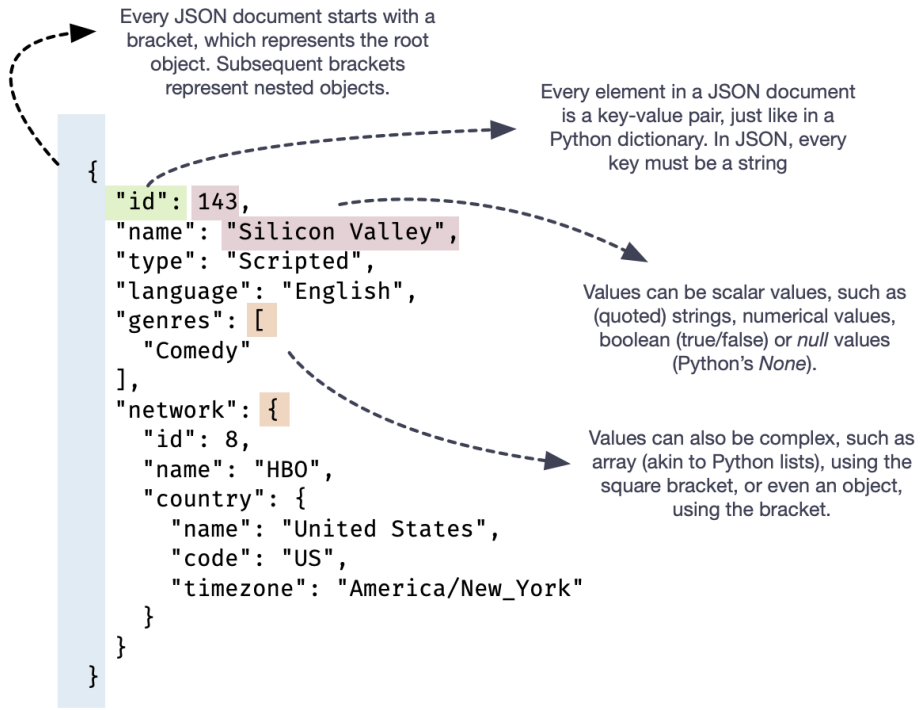
- **describe()** shows a summary of the data statistics, including central tendency and dispersion of the dataset's distribution:

```
flights.describe()
```

	Month	Day	Weekday	AirTime	Distance	ArrivalDelay	Diverted	Cancelled
count	58492.000000	58492.000000	58492.000000	57474.000000	58492.000000	57474.000000	58492.000000	58492.000000
mean	6.220646	15.702096	3.926862	115.928576	872.900072	5.812315	0.002342	0.015062
std	3.358484	8.760846	1.995777	71.679591	624.996805	38.411948	0.048340	0.121800
min	1.000000	1.000000	1.000000	8.000000	67.000000	-60.000000	0.000000	0.000000
25%	3.000000	8.000000	2.000000	61.000000	391.000000	-12.000000	0.000000	0.000000
50%	6.000000	16.000000	4.000000	97.000000	690.000000	-4.000000	0.000000	0.000000
75%	9.000000	23.000000	6.000000	152.000000	1199.000000	9.000000	0.000000	0.000000
max	12.000000	31.000000	7.000000	577.000000	4502.000000	1185.000000	1.000000	1.000000

JSON

- **J**ava**S**cript **O**bject **N**otation
- Textual representation widely used for data exchange between applications
- Built on two structures:
  - Objects (a collection of key/value pairs)
  - Arrays (an ordered list of values)
- Primitive types: Numbers, Strings, Booleans, null
- Syntax rules:
  - Data is stored in key/value pairs (as in dictionary)
  - Keys must be strings
  - Allows only double-quoted strings



Loading Data from JSON

Northeastern University

- You can load a JSON file into a DataFrame simply by calling **pd.read_json()**

```
population = pd.read_json('worldpopulation.json')
```

```
population.head()
```

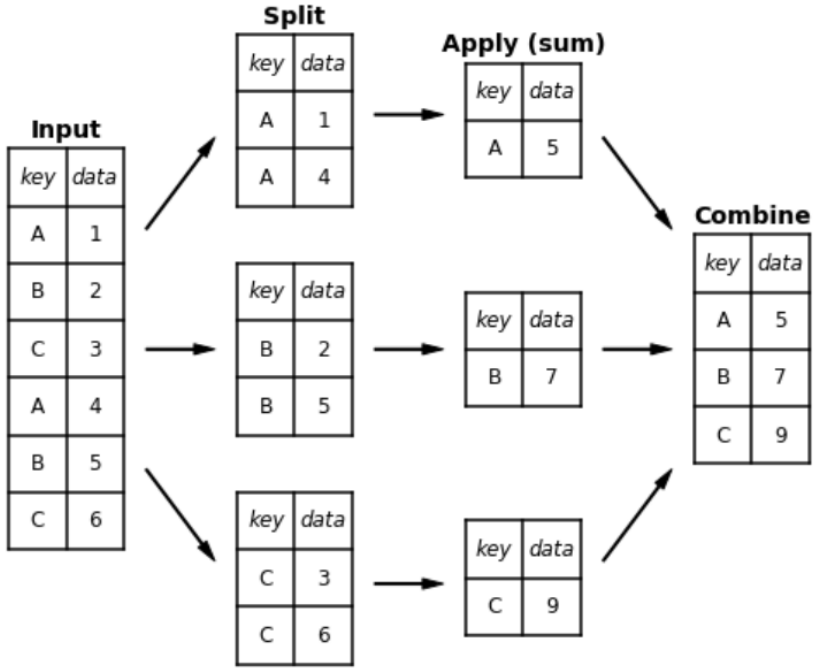
	Rank	country	population	World
0	1	China	1388232693	0.185
1	2	India	1342512706	0.179
2	3	U.S.	326474013	0.043
3	4	Indonesia	263510146	0.035
4	5	Brazil	211243220	0.028


```
[{
    "Rank": "1",
    "country": "China",
    "population": "1388232693",
    "World": "0.185"
}, {
    "Rank": "2",
    "country": "India",
    "population": "1342512706",
    "World": "0.179"
}, {
    "Rank": "3",
    "country": "U.S.",
    "population": "326474013",
    "World": "0.043"
}, …
]
```

Grouping

Northeastern University

- Grouping allows you to group rows that have the same values into summary rows
- For example, it allows you to write queries such as “find the number of customers in each country” or “find the airline with the highest number of flight cancels”
- Group operations consists of the following steps:
  - **Split** the data into groups based on a specified key
  - **Apply** a function to each group independently
  - **Combine** the results into a new DataFrame



Grouping

Northeastern University

- **groupby(*by*)** gets a column or a list of columns that determine the grouping
- It returns a DataFrameGroupBy object on which you can apply various aggregate functions such as sum(), count(), or mean()

```
flights.groupby('Airline').mean()
```

Airline	Month	Day	Weekday	AirTime	Distance	ArrivalDelay	Diverted	Cancelled
AA	6.661461	15.570562	3.986966	144.259404	1114.347865	5.542661	0.002921	0.017303
AS	6.197917	15.247396	3.941406	147.845052	1065.884115	-0.833333	0.000000	0.000000
B6	6.368324	15.788214	4.040516	209.412963	1771.882136	8.692593	0.003683	0.001842
DL	6.277332	15.721158	3.913310	115.334187	866.448448	0.339691	0.002264	0.003585
EV	6.003926	15.866166	3.927450	68.964016	460.237453	7.034580	0.002561	0.024923
F9	6.401670	15.914958	4.007593	127.592337	969.593014	13.630651	0.001519	0.007593

Column Indexing

Northeastern University

- You can select a particular column or columns from the GroupBy object:

```
flights.groupby('Airline')['ArrivalDelay'].mean()
```

```
Airline
AA     5.542661
AS    -0.833333
B6     8.692593
DL     0.339691
EV     7.034580
F9    13.630651
HA     4.972973
MQ     6.860591
NK    18.436070
OO     7.593463
UA     7.765755
US     1.681105
VX     5.348884
WN     6.397353
Name: ArrivalDelay, dtype: float64
```

Size of Groups

Northeastern University

- Use the **size()** method to get the number of objects in each group
  - It returns a Series and not a DataFrame like count(), since the size is the same for all columns

```
flights.groupby('Airline').size()
```

```
Airline
AA     8900
AS      768
B6      543
DL    10601
EV     5858
F9     1317
HA      112
MQ     3471
NK     1516
OO     6588
UA     7792
US     1615
VX      993
WN     8418
dtype: int64
```

Grouping by Multiple Columns

Northeastern University

- Grouping by multiple columns forms a hierarchical index:

```
flights.groupby(['Airline', 'Origin']).mean()
```

Airline	Origin	Month	Day	Weekday	AirTime	Distance	ArrivalDelay	Diverted	Cancelled
AA	ATL	7.596567	15.515021	3.789700	118.227074	809.781116	4.292576	0.004292	0.012876
	DEN	7.114155	15.525114	4.223744	116.037383	891.310502	9.485981	0.004566	0.018265
	DFW	6.193460	15.353969	4.000499	127.703221	959.727409	5.305215	0.001997	0.021468
	IAH	7.112245	15.341837	4.102041	97.098446	702.127551	10.813472	0.000000	0.015306
	LAS	6.828877	16.427807	4.010695	148.951087	1221.783422	7.907609	0.008021	0.008021
...	...	...	...	...	...	...	...	...	...
WN	LAS	6.327425	15.840473	3.905958	107.181458	835.102905	6.851760	0.003447	0.003447
	LAX	6.298678	15.715419	3.787665	104.938238	800.751542	11.192552	0.001762	0.028194
	MSP	6.071730	15.371308	3.789030	89.550847	607.059072	-0.271186	0.000000	0.004219
	PHX	6.151392	15.852668	3.860789	106.015743	818.176914	6.165015	0.001740	0.003480
	SFO	6.191011	15.038202	3.788764	93.735714	667.546067	4.473810	0.000000	0.056180

114 rows × 8 columns

Custom Aggregation

Northeastern University

- You can apply a custom aggregate function on the groups by calling the **agg**() method
- Its argument is a function that takes a Series object and returns a scalar value

```
flights.groupby('Airline')['ArrivalDelay'].agg(lambda x: x.max() - x.min())
```

```
Airline
AA     918.0
AS     401.0
B6     382.0
DL     798.0
EV     708.0
F9     882.0
HA     342.0
MQ     394.0
NK     513.0
OO     768.0
UA    1243.0
US     482.0
VX     287.0
WN     545.0
Name: ArrivalDelay, dtype: float64
```

Transformation

Northeastern University

- Allows you to change the values in a group based on the group properties
- Its argument is a function that takes a Series object and returns a Series object
- For example, we can center the data by subtracting the group-wise mean:

```
flights.groupby('Airline')['ArrivalDelay'].transform(lambda x: x - x.mean())
```

```
0         58.602647
1        -20.765755
2         28.139409
3        -12.542661
4         32.602647
             ...
58487    -24.542661
58488     -9.630651
58489    -12.593463
58490     27.602647
58491     -8.593463
Name: ArrivalDelay, Length: 58492, dtype: float64
```

Filtering

Northeastern University

- A filtering operation allows you to drop data based on the group properties
- Its argument is a function that takes a DataFrame object and returns True or False
- For example, we can find the airlines that had more than 100 cancellations

```
flights.groupby('Airline').filter(lambda df: df['Cancelled'].sum() > 100)
```

	Month	Day	Weekday	Airline	Origin	Dest	AirTime	Distance	ArrivalDelay	Diverted
2	1	1	4	MQ	DFW	VPS	85.0	641	35.0	0
3	1	1	4	AA	DFW	DCA	126.0	1192	-7.0	0
6	1	1	4	AA	DFW	MSY	64.0	447	83.0	0
8	1	1	4	AA	ORD	STL	44.0	258	-5.0	0
10	1	1	4	MQ	DFW	DRO	104.0	674	28.0	0
...	...	...	...	...	...	...	...	...	...	...
58484	12	31	4	MQ	ORD	DSM	57.0	299	-7.0	0
58486	12	31	4	EV	DFW	LFT	52.0	351	14.0	0
58487	12	31	4	AA	SFO	DFW	166.0	1464	-19.0	0
58489	12	31	4	OO	SFO	SBA	46.0	262	-5.0	0
58491	12	31	4	OO	SFO	BOI	73.0	522	-1.0	0

24817 rows × 11 columns

Filtering

Northeastern University

- filter() returns a copy of the DataFrame excluding the filtered elements
- In order to get only the groups, you need to apply another groupby() on the result

```
result = flights.groupby('Airline').filter(lambda df: df['Cancelled'].sum() > 100)
result.groupby('Airline')['Cancelled'].sum()
```

```
Airline
AA    154
EV    146
MQ    152
OO    142
Name: Cancelled, dtype: int64
```

Class Exercise

Northeastern University

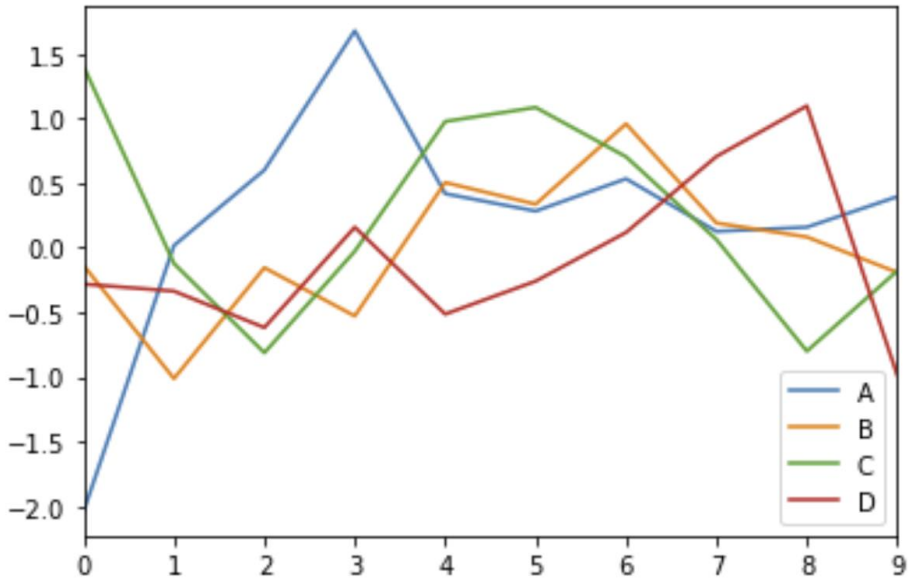
- In the animals DataFrame:
  - Display the number of animals from each type (dogs, cats and fish)
  - For each type of animal, show the heaviest animal
  - Display the animal types whose mean weight is greater than 10

Plotting

Northeastern University

- The DataFrame's **plot()** method is a convenience method to plot all the columns with their labels

```
df = pd.DataFrame(np.random.randn(10, 4), columns=list('ABCD'))
df.plot();
```



Customizing Plots

Northeastern University

- You can customize these plots by using one of the following arguments:
  - These arguments are passed through to the respective matplotlib plotting function

Argument	Description
kind	Can be 'line', 'bar', 'barh', 'hist', 'box', 'kde', 'density', 'area', 'pie'
ax	matplotlib axes to plot on
figsize	Size of figure to create as tuple
use_index	Use the object index for tick labels
title	Title to use for the plot
grid	Axis grid lines
legend	Add a subplot legend (True by default)
style	Style string, like 'ko--', to be passed to matplotlib (one per column)
logx	Use log scaling on the X axis
logy	Use log scaling on the Y axis

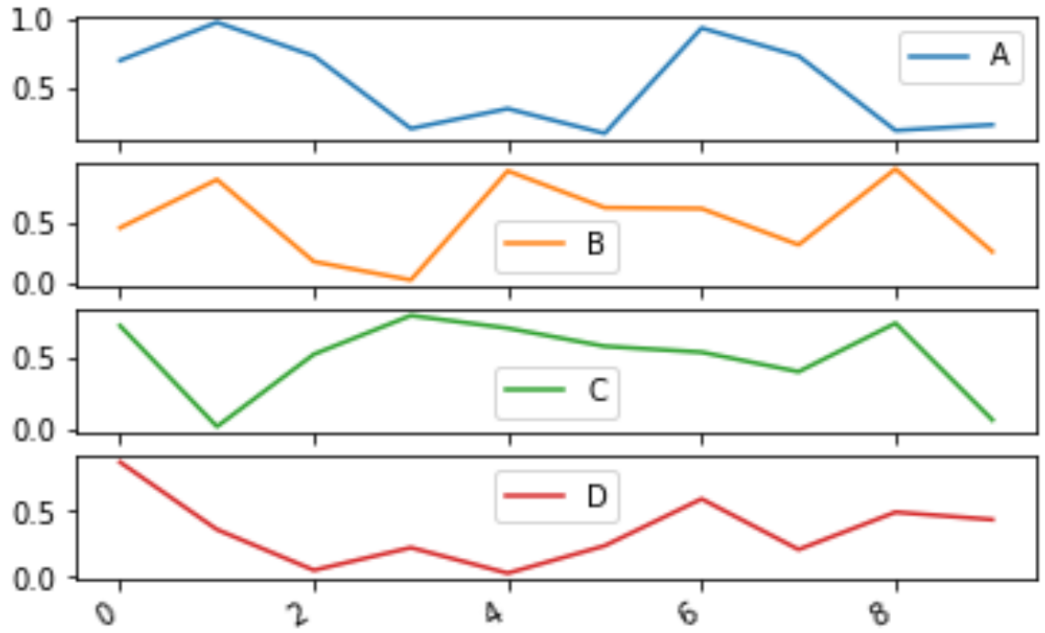
Argument	Description
xticks	Values to use for X axis ticks
yticks	Values to use for Y axis ticks
xlim,	X axis limits (e.g. [0, 10])
ylim	Y axis limits
rot	Rotation of tick labels (0 through 360)
fontsize	Font size for xticks and yticks
colormap	Colormap to select colors from
xerr	X axis error data
yerr	Y axis error data
label	label argument to provide to plot
**kwds	Options to pass to matplotlib plotting method

Customizing Plots

Northeastern University

- For example, you can create a separate subplot for each column:

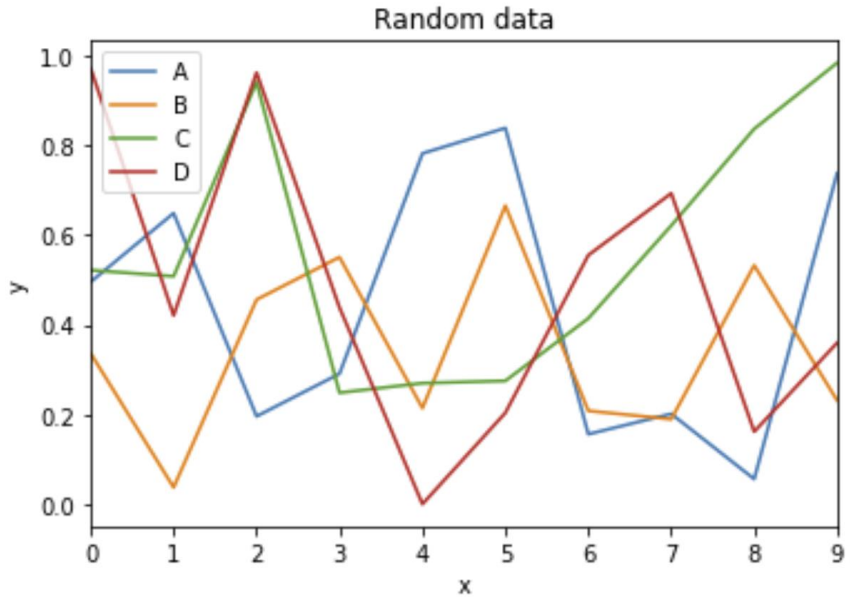
```
df = pd.DataFrame(np.random.rand(10, 4), columns=list('ABCD'))
df.plot(subplots=True);
```



Customizing Plots

- The plot() function returns an Axes object, which you can use to further customize the plot:

```
df = pd.DataFrame(np.random.rand(10, 4), columns=list('ABCD'))
ax = df.plot()
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Random data');
```



Plot Types

Northeastern University

- Other types of plots can be created by passing the *kind* argument to df.plot(), or by calling df.plot.*fig_type*(), e.g., df.plot.hist()
- Supported figure types:

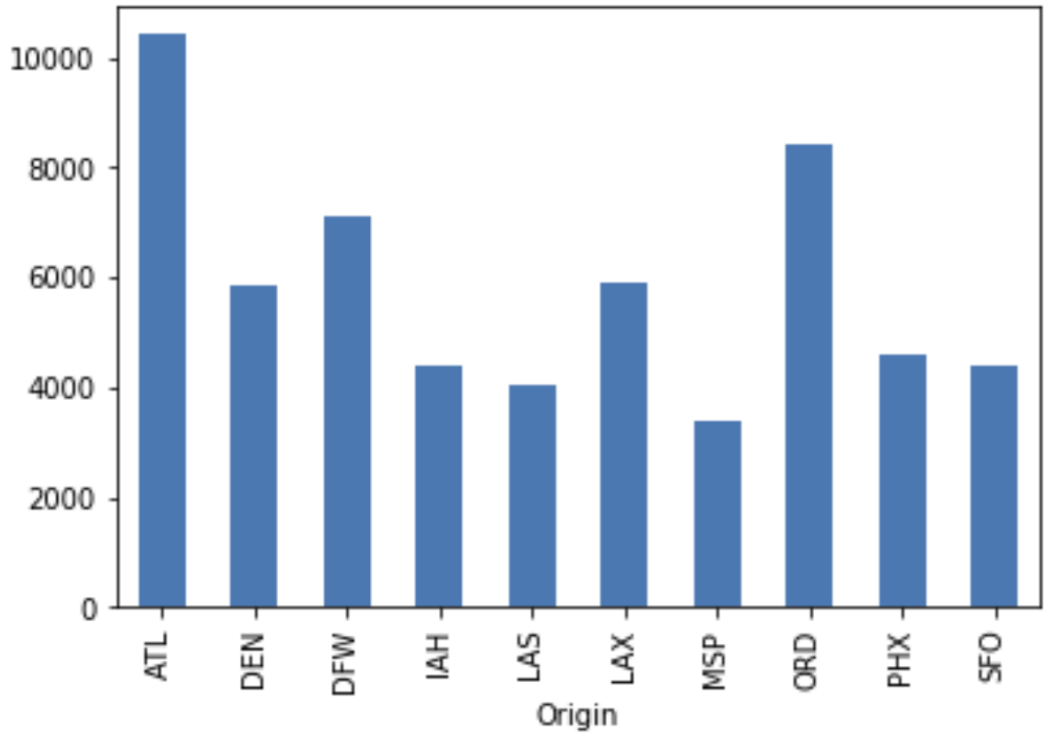
kind	Description
line	line plot (default)
bar	vertical bar plot
barh	horizontal bar plot
hist	histogram
box	boxplot
kde, density	Kernel Density Estimation plot
area	area plot
pie	pie plot
scatter	scatter plot

Plotting

Northeastern University

- For example, let's create a bar plot of the number of flights from each origin:

```
flights.groupby('Origin').size().plot.bar();
```



Combining DataFrames

Northeastern University

- Many interesting studies of data come from combining different data sources
- Pandas allows you to combine data frames in different ways:
  - **pd.concat()** stacks together data frames along an axis
  - **pd.merge()** joins rows in the data frames based on one or more keys
    - Similar to SQL JOIN operation

Concatenating DataFrames

Northeastern University

- **pd.concat**(*objects*, *axis=0*) concatenates DataFrames or Series along a particular axis
- For example, we can use it to add new rows to an existing DataFrame:

```
new_states = pd.DataFrame({
    'capital': ['Trenton', 'Lansing'],
    'population': [8882190, 9986857],
    'area': [8723, 96714]
}, index=['New Jersey', 'Michigan'])

pd.concat((states, new_states))
```

	capital	population	area
California	Sacramento	39512223	163695
Massachusetts	Boston	6892503	10554
Texas	Austin	28995881	268596
New York	Albany	19453561	54555
Florida	Tallahassee	21477737	65758
New Jersey	Trenton	8882190	8723
Michigan	Lansing	9986857	96714

Merging DataFrames

Northeastern University

pandas.merge(*left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes='_x', '_y', copy=True, indicator=False, validate=None*)

- Merges DataFrames with a database-style join
- Arguments:
  - **left** – DataFrame
  - **right** – DataFrame to merge with
  - **how** – type of merge to be performed:
    - inner (default): use intersection of keys from both frames, similar to a SQL inner join
    - left: use only keys from left frame, similar to a SQL left outer join
    - right: use only keys from right frame, similar to a SQL right outer join
    - outer: use union of keys from both frames, similar to a SQL full outer join
  - **on** – column names to join on. These must be found in both DataFrames.

Merging DataFrames Example

Northeastern University

```
df1 = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
                    'A': ['A0', 'A1', 'A2'],
                    'B': ['B0', 'B1', 'B2']})
df1
```

	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2

```
df2 = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
                    'C': ['C0', 'C1', 'C2'],
                    'D': ['D0', 'D1', 'D2']})
df2
```

	key	C	D
0	K0	C0	D0
1	K1	C1	D1
2	K2	C2	D2

```
pd.merge(df1, df2)
```

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2

Specifying the Merge Key

Northeastern University

- By default pd.merge() looks for one or more matching column names between the two DataFrames, and uses this as the key
- You can also explicitly specify the key columns using *on*, *left_on* and *right_on*

```
df1 = pd.DataFrame({'lkey': ['K0', 'K1', 'K2'],
                    'A': ['A0', 'A1', 'A2'],
                    'B': ['B0', 'B1', 'B2']})
df2 = pd.DataFrame({'rkey': ['K0', 'K1', 'K2'],
                    'C': ['C0', 'C1', 'C2'],
                    'D': ['D0', 'D1', 'D2']})
pd.merge(df1, df2, left_on='lkey', right_on='rkey')
```

	lkey	A	B	rkey	C	D
0	K0	A0	B0	K0	C0	D0
1	K1	A1	B1	K1	C1	D1
2	K2	A2	B2	K2	C2	D2

Merging on an Index

Northeastern University

- You can use the index as the merge key by specifying *left_index* and/or *right_index*:

```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                    'B': ['B0', 'B1', 'B2']},
                   index=['K0', 'K1', 'K2'])
df2 = pd.DataFrame({'C': ['C0', 'C1', 'C2'],
                    'D': ['D0', 'D1', 'D2']},
                   index=['K0', 'K1', 'K2'])
pd.merge(df1, df2, left_index=True, right_index=True)
```

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	C1	D1
K2	A2	B2	C2	D2

Time Series

Northeastern University

- Pandas contains extensive capabilities and features for working with time series data
- Pandas provides a **Timestamp** type, which combines the ease-of-use of Python's datetime with the efficient storage and vectorized interface of numpy.datetime64
- Assume that we have the following date objects represented by strings:

```
df = pd.DataFrame({'Date': ['8/3/2020','5/3/2020','10/3/2020','12/3/2020'],
                   'A': ['A0', 'A1', 'A2', 'A3'],
                   'B': ['B0', 'B1', 'B2', 'B3']})
```

```
df.sort_values('Date')
```

	Date	A	B
2	10/3/2020	A2	B2
3	12/3/2020	A3	B3
1	5/3/2020	A1	B1
0	8/3/2020	A0	B0

Time Series

Northeastern University

- To convert a Series or a list of strings representing dates into date time objects, you can use the function **pd.to_datetime()**

```
df['Date'] = pd.to_datetime(df['Date'])
df.sort_values('Date')
```

	Date	A	B
1	2020-05-03	A1	B1
0	2020-08-03	A0	B0
2	2020-10-03	A2	B2
3	2020-12-03	A3	B3

Time Series

Northeastern University

- You can specify the format of the dates using the **format** argument:

```
df['Date'] = pd.to_datetime(df['Date'], format='%d/%m/%Y')
df.sort_values('Date')
```

	Date	A	B
1	2020-03-05	A1	B1
0	2020-03-08	A0	B0
2	2020-03-10	A2	B2
3	2020-03-12	A3	B3

Indexing by Time

Northeastern University

- Pandas time series tools really become useful when you index data by timestamps
- To take advantage of them, you need to define the table's index as DatetimeIndex

```
index = pd.DatetimeIndex(['2020-07-01', '2020-08-01', '2021-07-01', '2021-08-01'])
df = pd.DataFrame({'A': [10, 20, 30, 40],
                   'B': [100, 200, 300, 400]}, index=index)
df
```

	A	B
2020-07-01	10	100
2020-08-01	20	200
2021-07-01	30	300
2021-08-01	40	400

Indexing by Time

Northeastern University

- We can now can slice the table rows by using a data range:

```
df['2020-01-01': '2021-01-01']
```

	A	B
2020-07-01	10	100
2020-08-01	20	200

- Or use the year as an index to obtain a slice of all data from that year:

```
df['2020']
```

	A	B
2020-07-01	10	100
2020-08-01	20	200

Grouping by Time

Northeastern University

- **pd.Grouper** allows you to group data by time frequencies
  - The argument *freq* in its constructor specifies the frequency
- For example, we can compute the mean of the data for every year:

```
df.groupby(pd.Grouper(freq='Y')).mean()
```

	A	B
2020-12-31	15	150
2021-12-31	35	350