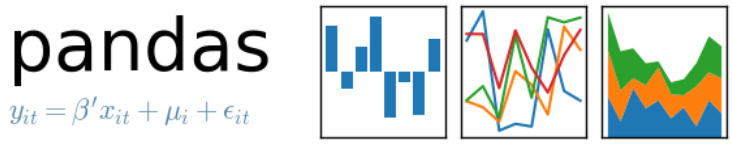
CSCI 303

Introduction to Data Science

9 - pandas basics



This Lecture

- Learn pandas basics

The obligatory setup code...

In [43]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn as sk
import sklearn.datasets

%matplotlib inline
```

pandas

Python toolkit for data analysis

- provides Series and DataFrame data structures
- DataFrame type inspired by R
- designed to interact with the whole Python data science stack
- eases many of the data science tasks, particularly data "wrangling"

Series

A one-dimensional array-like object:

- contains a sequence of values of one particular type
- has an associated array of *index* labels
  - labels do not have to be integers
  - labels do not have to be unique
  - labels do not have to be sequential

Like a NumPy array, a Series can be constructed from any iterable:

In [44]:
```
from pandas import Series

s = Series([42, 17, 99])
s
```

Out[44]:
```
0    42
1    17
2    99
dtype: int64
```

The *index* is shown on the left

- default: RangeIndex (representing sequential integers)
- access index via `index` property of the Series object

In [45]:
```
s.index
```

Out[45]:
```
RangeIndex(start=0, stop=3, step=1)
```

There is also a `values` property:

In [46]:
```
s.values
# shows the values in the Series
```

Out[46]:
```
array([42, 17, 99])
```

Things get interesting when you use *labels* for the index:

In [47]:
```
s = Series([42, 17, 99], index=['apple', 'pear', 'orange'])
s
```

Out[47]:
```
apple     42
pear      17
orange    99
dtype: int64
```

Like a dictionary:

- associate values with labels
- retrieve values via [ ] operator

Unlike a dictionary:

- retain original order
- labels can duplicate

In [48]:
```
s2 = Series([42, 17, 99, 3.1415], index=['apple', 'pear', 'orange', 'apple'])
s2
```

Out[48]:
```
apple     42.0000
pear      17.0000
orange    99.0000
apple      3.1415
dtype: float64
```

In [49]:
```
s2['orange']
```

Out[49]:
```
99.0
```

In [50]:
```
#fields = ['orange', 'pear']
s2[['orange', 'pear']] # can access more than one value at a time
```

Out[50]:
```
orange    99.0
pear      17.0
dtype: float64
```

In [51]:
```
s2['apple'] # when duplicate labels are in the Series, both values are shown that correspond with each label
```

Out[51]:
```
apple    42.0000
apple     3.1415
dtype: float64
```

In [52]:
```
test = Series([1,2,3],index=['foo',17,True])
test
```

Out[52]:
```
foo     1
17      2
True    3
dtype: int64
```

Note the last two lookups resulted in Series objects.

You can apply math and other NumPy-like operations:

In [53]:
```
s2
```

Out[53]:
```
apple     42.0000
pear      17.0000
orange    99.0000
apple      3.1415
dtype: float64
```

In [54]:
```
s2 = s2 * 2
```

In [55]:
```
np.cos(s2)
```

Out[55]:
```
apple    -0.680023
pear     -0.848570
orange   -0.996829
apple     1.000000
dtype: float64
```

Data aligns by label in arithmetic operations:

In [56]:
```
s3 = Series([1, 2, 3, 4], ['a', 'b', 'c', 'd'])
s4 = Series([5, 6, 7, 8, 9], ['d', 'b', 'a', 'e', 'd'])
s3 + s4
```

Out[56]:
```
a     8.0
b     8.0
c     NaN
d     9.0
d    13.0
e     NaN
dtype: float64
```

In [57]:
```
s5 = Series(['hello', 'goodbye', np.NaN], index=['a','b','c'])
s5
```

Out[57]:
```
a      hello
b    goodbye
c        NaN
dtype: object
```

Note the unmatched labels turned into NaNs - pandas notation for missing data.

Series objects can also be *named*, via the `name` property:

In [58]:
```
s2.name = 'tonnes'
s2
```

Out[58]:
```
apple      84.000
pear       34.000
orange    198.000
apple       6.283
Name: tonnes, dtype: float64
```

The index can also be named:

In [59]:
```
s2.index.name = 'fruit'
#s2['orange']
s2
```

Out[59]:
```
fruit
apple      84.000
pear       34.000
orange    198.000
apple       6.283
Name: tonnes, dtype: float64
```

DataFrame

A data structure which functions much like a database table

- ordered collection of columns, each of a specific type

- column index labels the columns, similar to attribute names
- row index labels rows, similar to a primary key

However, more complex than a database table (and more powerful!)

You can make a DataFrame object from a dictionary object:

In [60]:
```
from pandas import DataFrame

df = DataFrame(
    {'fruit' : ['apple', 'orange', 'peach', 'apple'],
     'tonnes' : [42, 17, 99, 3.1415],
     'type' : ['pome', 'citrus', 'drupe', 'pome']})

df.index = ['crate a', 'crate b', 'crate w', 'crate f']
df
print(df)
print(df[:2][['fruit','tonnes']]) # shows the first two crates in the dataframe
```

```
          fruit   tonnes    type
crate a   apple  42.0000    pome
crate b  orange  17.0000  citrus
crate w   peach  99.0000   drupe
crate f   apple   3.1415    pome
          fruit  tonnes
crate a   apple    42.0
crate b  orange    17.0
```

In [61]:
```
# use df.head() to show the entire dataframe as well
df.head(2)
```

Out[61]:

	fruit	tonnes	type
crate a	apple	42.0	pome
crate b	orange	17.0	citrus

...although mostly we'll be getting DataFrames in other ways, such as from external sources.

DataFrame objects have much of the same extensible naming/indexing as Series objects:

In [62]:
```
df.index = ['crate 1', 'crate 2', 'crate 16', 'crate 11']
df
```

Out[62]:

	fruit	tonnes	type
crate 1	apple	42.0000	pome
crate 2	orange	17.0000	citrus
crate 16	peach	99.0000	drupe
crate 11	apple	3.1415	pome

In [63]:
```
df.index.name = 'location'
df
```

Out[63]:

location	fruit	tonnes	type
crate 1	apple	42.0000	pome
crate 2	orange	17.0000	citrus
crate 16	peach	99.0000	drupe
crate 11	apple	3.1415	pome

You access columns by name, usign either [ ] or the . operator:

In [64]:
```
df['fruit']  # or df.fruit
```

Out[64]:
```
location
crate 1       apple
crate 2      orange
crate 16      peach
crate 11      apple
Name: fruit, dtype: object
```

In [65]:
```
df[['tonnes', 'fruit']]
mySeries = df[:1]
mySeries
```

Out[65]:

location	fruit	tonnes	type
crate 1	apple	42.0	pome

However, note that slicing notation applies to rows:

In [66]:
```
df[1:3]
```

Out[66]:

location	fruit	tonnes	type
crate 2	orange	17.0	citrus
crate 16	peach	99.0	drupe

You can more precisely access rows by label or position using the `loc` and `iloc` special operators (*not methods!*):

In [67]:
```
df.loc['crate 16', ['fruit','tonnes']]
```

Out[67]:
```
fruit     peach
tonnes     99.0
Name: crate 16, dtype: object
```

In [68]:
```
df.loc[:'crate 16', ['type', 'tonnes']]
```

Out[68]:

location	type	tonnes
crate 1	pome	42.0
crate 2	citrus	17.0
crate 16	drupe	99.0

In [69]:
```
df.iloc[1:3,1:2]
```

Out[69]:

location	tonnes
crate 2	17.0
crate 16	99.0

In [70]:
```
df.iloc[3]
```

Out[70]:
```
fruit      apple
tonnes    3.1415
type        pome
Name: crate 11, dtype: object
```

There's also Boolean indexing:

In [71]:
```
df[df['fruit']=='apple']
```

Out[71]:

location	fruit	tonnes	type
crate 1	apple	42.0000	pome
crate 11	apple	3.1415	pome

In [72]:
```
df[df.tonnes > 20]
```

Out[72]:

location	fruit	tonnes	type
crate 1	apple	42.0	pome
crate 16	peach	99.0	drupe

Confused yet?

We'll explore these further as needed. Don't forget the pandas documentation under the Help menu in your notebook!

Also, here's a "cheat sheet" (https://github.com/pandas-dev/pandas/blob/master/doc/cheatsheet/Pandas_Cheat_Sheet.pdf).

The Boston Housing Dataset (REMOVED) - Now using California Housing Dataset

A well known and heavily studied dataset for statistical inference.

Available in the scikit-learn package, or many sources online.

In [73]:
```
from sklearn.datasets import fetch_california_housing
raw = fetch_california_housing()
```

In [74]:
```
raw.keys()
```

Out[74]:
```
dict_keys(['data', 'target', 'frame', 'target_names', 'feature_names', 'DESCR'])
```

In [75]:
```
1 print(raw.DESCR)
```

```
.. _california_housing_dataset:

California Housing dataset
--------------------------

**Data Set Characteristics:**

    :Number of Instances: 20640

    :Number of Attributes: 8 numeric, predictive attributes and the target

    :Attribute Information:
        - MedInc        median income in block group
        - HouseAge      median house age in block group
        - AveRooms      average number of rooms per household
        - AveBedrms     average number of bedrooms per household
        - Population    block group population
        - AveOccup      average number of household members
        - Latitude      block group latitude
        - Longitude     block group longitude

    :Missing Attribute Values: None

This dataset was obtained from the StatLib repository.
https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html (https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html)

The target variable is the median house value for California districts,
expressed in hundreds of thousands of dollars ($100,000).

This dataset was derived from the 1990 U.S. census, using one row per census
block group. A block group is the smallest geographical unit for which the U.S.
Census Bureau publishes sample data (a block group typically has a population
of 600 to 3,000 people).

An household is a group of people residing within a home. Since the average
number of rooms and bedrooms in this dataset are provided per household, these
columns may take surpinsingly large values for block groups with few households
and many empty houses, such as vacation resorts.

It can be downloaded/loaded using the
:func:`sklearn.datasets.fetch_california_housing` function.

.. topic:: References

    - Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,
      Statistics and Probability Letters, 33 (1997) 291-297
```

We can view the raw data and target arrays...

In [76]:
```
1 raw.data
```

Out[76]:
```
array([[   8.3252    ,   41.        ,    6.98412698, ...,    2.55555556,
          37.88      , -122.23      ],
       [   8.3014    ,   21.        ,    6.23813708, ...,    2.10984183,
          37.86      , -122.22      ],
       [   7.2574    ,   52.        ,    8.28813559, ...,    2.80225989,
          37.85      , -122.24      ],
       ...,
       [   1.7       ,   17.        ,    5.20554273, ...,    2.3256351 ,
          39.43      , -121.22      ],
       [   1.8672    ,   18.        ,    5.32951289, ...,    2.12320917,
          39.43      , -121.32      ],
       [   2.3886    ,   16.        ,    5.25471698, ...,    2.61698113,
          39.37      , -121.24      ]])
```

In [77]:
```
1 raw.target
```

Out[77]:
```
array([4.526, 3.585, 3.521, ..., 0.923, 0.847, 0.894])
```

Instead, let's load the data into a DataFrame where we can explore it a bit more easily.

Along the way, we'll explore some of the DataFrame object's interface.

In [78]:
```
cali = DataFrame(raw.data, columns=raw.feature_names)
```

In [79]:
```
cali
```

Out[79]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25
...	...	...	...	...	...	...	...	...
20635	1.5603	25.0	5.045455	1.133333	845.0	2.560606	39.48	-121.09
20636	2.5568	18.0	6.114035	1.315789	356.0	3.122807	39.49	-121.21
20637	1.7000	17.0	5.205543	1.120092	1007.0	2.325635	39.43	-121.22
20638	1.8672	18.0	5.329513	1.171920	741.0	2.123209	39.43	-121.32
20639	2.3886	16.0	5.254717	1.162264	1387.0	2.616981	39.37	-121.24

20640 rows × 8 columns

Adding/deleting a column is simple:

In [80]:
```
cali['Target'] = raw.target
#del cali['Target']
cali[:10]
```

Out[80]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	Target
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422
5	4.0368	52.0	4.761658	1.103627	413.0	2.139896	37.85	-122.25	2.697
6	3.6591	52.0	4.931907	0.951362	1094.0	2.128405	37.84	-122.25	2.992
7	3.1200	52.0	4.797527	1.061824	1157.0	1.788253	37.84	-122.25	2.414
8	2.0804	42.0	4.294118	1.117647	1206.0	2.026891	37.84	-122.26	2.267
9	3.6912	52.0	4.970588	0.990196	1551.0	2.172269	37.84	-122.25	2.611

Basic Statistics

pandas provides the `describe` function (similar to R's `summary`):

In [81]:
```
cali.describe()
```

Out[81]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	Target
count	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	3.870671	28.639486	5.429000	1.096675	1425.476744	3.070655	35.631861	-119.569704	2.068558
std	1.899822	12.585558	2.474173	0.473911	1132.462122	10.386050	2.135952	2.003532	1.153956
min	0.499900	1.000000	0.846154	0.333333	3.000000	0.692308	32.540000	-124.350000	0.149990
25%	2.563400	18.000000	4.440716	1.006079	787.000000	2.429741	33.930000	-121.800000	1.196000
50%	3.534800	29.000000	5.229129	1.048780	1166.000000	2.818116	34.260000	-118.490000	1.797000
75%	4.743250	37.000000	6.052381	1.099526	1725.000000	3.282261	37.710000	-118.010000	2.647250
max	15.000100	52.000000	141.909091	34.066667	35682.000000	1243.333333	41.950000	-114.310000	5.000010

pandas has other convenience methods. How about pairwise correlations in the data?

In [82]:
```
1 cali.corr()
```

Out[82]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	Target
MedInc	1.000000	-0.119034	0.326895	-0.062040	0.004834	0.018766	-0.079809	-0.015176	0.688075
HouseAge	-0.119034	1.000000	-0.153277	-0.077747	-0.296244	0.013191	0.011173	-0.108197	0.105623
AveRooms	0.326895	-0.153277	1.000000	0.847621	-0.072213	-0.004852	0.106389	-0.027540	0.151948
AveBedrms	-0.062040	-0.077747	0.847621	1.000000	-0.066197	-0.006181	0.069721	0.013344	-0.046701
Population	0.004834	-0.296244	-0.072213	-0.066197	1.000000	0.069863	-0.108785	0.099773	-0.024650
AveOccup	0.018766	0.013191	-0.004852	-0.006181	0.069863	1.000000	0.002366	0.002476	-0.023737
Latitude	-0.079809	0.011173	0.106389	0.069721	-0.108785	0.002366	1.000000	-0.924664	-0.144160
Longitude	-0.015176	-0.108197	-0.027540	0.013344	0.099773	0.002476	-0.924664	1.000000	-0.045967
Target	0.688075	0.105623	0.151948	-0.046701	-0.024650	-0.023737	-0.144160	-0.045967	1.000000

We can take sums, means, standard deviations, etc. by row or column:

In [83]:
```
1 cali.mean()
```

Out[83]:
```
MedInc            3.870671
HouseAge         28.639486
AveRooms          5.429000
AveBedrms         1.096675
Population     1425.476744
AveOccup          3.070655
Latitude         35.631861
Longitude      -119.569704
Target            2.068558
dtype: float64
```

In [84]:
```
1 cali.sum(axis=1)[:10]
```

Out[84]:
```
0      302.064692
1     2358.846259
2      486.552242
3      544.094456
4      549.412602
5      395.338981
6     1076.252773
7     1137.771604
8     1175.366055
9     1533.025253
dtype: float64
```

Next Time

Next lecture, we'll do some exploratory data analysis on the California housing set.