Lecture 9 Introduction to Pandas

[Pandas--*Python Data Analysis Library* (https://pandas.pydata.org/)](https://pandas.pydata.org/) provides the high-performance, easy-to-use data structures and data analysis tools in Python, which is very useful in Data Science. In our lectures, we only focust on the [elementary usages (https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html)](https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html).

In [1]:
```python
import pandas as pd
import numpy as np
```

In [ ]:
```python
pd.__version__
```

In [ ]:
```python
dir(pd)
```

Important Concepts: `Series` and `DataFrame`

In short, `Series` represents one variable (attributes) of the datasets, while `DataFrame` represents the whole tabular data (it also supports multi-index or tensor cases -- we will not discuss these cases here).

`Series` is Numpy 1d array-like, additionally featuring for "index" which denotes the sample name, which is also similar to Python built-in dictionary type.

In [ ]:
```python
s1 = pd.Series([2, 4, 6])
```

In [ ]:
```python
type(s1)
```

In [ ]:
```python
s1.index
```

In [2]:
```python
s2 = pd.Series([2, 4, 6],index = ['a','b','c'])
```

In [3]:
```python
s2
```

Out[3]:
```
a    2
b    4
c    6
dtype: int64
```

In [4]:
```python
s2_num = s2.values # change to Numpy -- can be view instead of copy if the elements a
re all numbers
s2_num
```

Out[4]:
```
array([2, 4, 6])
```

In [5]:
```python
np.shares_memory(s2_num,s2)
```

Out[5]:
```
True
```

In [6]:
```python
s2_num_copy = s2.to_numpy(copy = True) # more recommended in new version of Pandas --
can specify view/copy
np.shares_memory(s2_num_copy,s2)
```

Out[6]:
```
False
```

Selection by position -- similar to Numpy array!

```
In [7]: s2[0:2]
```

```
Out[7]: a    2
        b    4
        dtype: int64
```

Selection by index (label)

```
In [8]: s2['a']
        s2[['a','b']]
```

```
Out[8]: a    2
        b    4
        dtype: int64
```

`Series` and Python Dictionary

```
In [9]: population_dict = {'California': 38332521,
                           'Texas': 26448193,
                           'New York': 19651127,
                           'Florida': 19552860,
                           'Illinois': 12882135} # this is the built-in python dictionary
        population = pd.Series(population_dict) # initialize Series with dictionary
        population
```

```
Out[9]: California    38332521
        Texas         26448193
        New York      19651127
        Florida       19552860
        Illinois      12882135
        dtype: int64
```

```
In [10]: population_dict['Texas'] # key and value
```

```
Out[10]: 26448193
```

```
In [11]: area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
                      'Florida': 170312, 'Illinois': 149995}
         area = pd.Series(area_dict)
         area
```

```
Out[11]: California    423967
         Texas         695662
         New York      141297
         Florida       170312
         Illinois      149995
         dtype: int64
```

Create the pandas `DataFrame` from `Series` . Note that in Pandas, the row/column of `DataFrame` are termed as `index` and `columns` .

In [12]:
```python
states = pd.DataFrame({'population': population,
                       'area': area}) # variable names
states
```

Out[12]:

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

In [13]:
```python
type(states)
```

Out[13]:
```
pandas.core.frame.DataFrame
```

In [14]:
```python
states.index
```

Out[14]:
```
Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='object')
```

In [15]:
```python
states.columns
```

Out[15]:
```
Index(['population', 'area'], dtype='object')
```

In [16]:
```python
states['area']
```

Out[16]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

In [17]:
```python
states.area
```

Out[17]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

In [18]:
```python
type(states['area'])
```

Out[18]:
```
pandas.core.series.Series
```

In [19]:
```python
random = pd.DataFrame(np.random.rand(3, 2),columns=['foo', 'bar'],index=['a', 'b', 'c'])
random
```

Out[19]:

	foo	bar
a	0.541344	0.724318
b	0.913853	0.646869
c	0.037833	0.962765

In [20]:
```
random.T
```
Out[20]:

	a	b	c
foo	0.541344	0.913853	0.037833
bar	0.724318	0.646869	0.962765

Creating DataFrame from Files

In [21]:
```
house_price = pd.read_csv('kc_house_data.csv')
house_price
```
Out[21]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	(
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	(
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	0	(
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	0	(
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	0	(
...	...	...	...	...	...	...	...	...	...	..
21608	263000018	20140521T000000	360000.0	3	2.50	1530	1131	3.0	0	(
21609	6600060120	20150223T000000	400000.0	4	2.50	2310	5813	2.0	0	(
21610	1523300141	20140623T000000	402101.0	2	0.75	1020	1350	2.0	0	(
21611	291310100	20150116T000000	400000.0	3	2.50	1600	2388	2.0	0	(
21612	1523300157	20141015T000000	325000.0	2	0.75	1020	1076	2.0	0	(

21613 rows × 21 columns

In [22]:
```
house_price.shape # dimension of the data
```
Out[22]:
```
(21613, 21)
```

In [23]:
```
house_price.info() # basic dataset information
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21613 non-null  int64
 1   date           21613 non-null  object
 2   price          21613 non-null  float64
 3   bedrooms       21613 non-null  int64
 4   bathrooms      21613 non-null  float64
 5   sqft_living    21613 non-null  int64
 6   sqft_lot       21613 non-null  int64
 7   floors         21613 non-null  float64
 8   waterfront     21613 non-null  int64
 9   view           21613 non-null  int64
 10  condition      21613 non-null  int64
 11  grade          21613 non-null  int64
 12  sqft_above     21613 non-null  int64
 13  sqft_basement  21613 non-null  int64
 14  yr_built       21613 non-null  int64
 15  yr_renovated   21613 non-null  int64
 16  zipcode        21613 non-null  int64
 17  lat            21613 non-null  float64
 18  long           21613 non-null  float64
 19  sqft_living15  21613 non-null  int64
 20  sqft_lot15     21613 non-null  int64
dtypes: float64(5), int64(15), object(1)
memory usage: 3.5+ MB
```

In [24]:
```
house_price.head(3) # show the head lines
```

Out[24]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	...
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	...
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	0	0	...

3 rows × 21 columns

In [25]:
```
house_price.sample(5) # show the random samples
```

Out[25]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view
7169	2085200545	20140821T000000	180000.0	3	1.0	840	5700	1.0	0	(
7586	9352901085	20150204T000000	256000.0	3	1.0	1290	4720	1.0	0	(
18223	3307700405	20140723T000000	587100.0	2	1.0	1190	6967	1.0	0	(
18107	6675500112	20150414T000000	330000.0	3	1.0	960	7218	1.0	0	(
6672	2725069150	20140817T000000	710000.0	3	2.5	2830	9680	2.0	0	(

5 rows × 21 columns

In [26]:
```
house_price.describe() # descriptive statistics
```

Out[26]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wat
count	2.161300e+04	2.161300e+04	21613.000000	21613.000000	21613.000000	2.161300e+04	21613.000000	21613.(
mean	4.580302e+09	5.401822e+05	3.370842	2.114757	2079.899736	1.510697e+04	1.494309	0.(
std	2.876566e+09	3.673622e+05	0.930062	0.770163	918.440897	4.142051e+04	0.539989	0.(
min	1.000102e+06	7.500000e+04	0.000000	0.000000	290.000000	5.200000e+02	1.000000	0.(
25%	2.123049e+09	3.219500e+05	3.000000	1.750000	1427.000000	5.040000e+03	1.000000	0.(
50%	3.904930e+09	4.500000e+05	3.000000	2.250000	1910.000000	7.618000e+03	1.500000	0.(
75%	7.308900e+09	6.450000e+05	4.000000	2.500000	2550.000000	1.068800e+04	2.000000	0.(
max	9.900000e+09	7.700000e+06	33.000000	8.000000	13540.000000	1.651359e+06	3.500000	1.(

In [27]:
```
head = house_price.head()
head.to_csv('head.csv')
```

In [28]:
```
head.sort_values(by='price')
```

Out[28]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	0	0	...
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	...
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	0	0	...
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	...
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	0	0	...

5 rows × 21 columns

In [29]:
```
help(head.sort_values)
```

```
Help on method sort_values in module pandas.core.frame:

sort_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', na_position
='last', ignore_index=False, key: 'ValueKeyFunc' = None) method of pandas.core.fram
e.DataFrame instance
    Sort by the values along either axis.

    Parameters
    ----------
            by : str or list of str
                Name or list of names to sort by.

                - if `axis` is 0 or `'index'` then `by` may contain index
                  levels and/or column labels.
                - if `axis` is 1 or `'columns'` then `by` may contain column
                  levels and/or index labels.
    axis : {0 or 'index', 1 or 'columns'}, default 0
         Axis to be sorted.
    ascending : bool or list of bool, default True
         Sort ascending vs. descending. Specify list for multiple sort
         orders.  If this is a list of bools, must match the length of
         the by.
    inplace : bool, default False
         If True, perform operation in-place.
    kind : {'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'
         Choice of sorting algorithm. See also ndarray.np.sort for more
         information.  `mergesort` is the only stable algorithm. For
         DataFrames, this option is only applied when sorting on a single
         column or label.
    na_position : {'first', 'last'}, default 'last'
         Puts NaNs at the beginning if `first`; `last` puts NaNs at the
         end.
    ignore_index : bool, default False
         If True, the resulting axis will be labeled 0, 1, …, n - 1.

         .. versionadded:: 1.0.0

    key : callable, optional
        Apply the key function to the values
        before sorting. This is similar to the `key` argument in the
        builtin :meth:`sorted` function, with the notable difference that
        this `key` function should be *vectorized*. It should expect a
        ``Series`` and return a Series with the same shape as the input.
        It will be applied to each column in `by` independently.

        .. versionadded:: 1.1.0

    Returns
    -------
    DataFrame or None
        DataFrame with sorted values or None if ``inplace=True``.

    See Also
    --------
    DataFrame.sort_index : Sort a DataFrame by the index.
    Series.sort_values : Similar method for a Series.

    Examples
    --------
    >>> df = pd.DataFrame({
    ...     'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
    ...     'col2': [2, 1, 9, 8, 7, 4],
    ...     'col3': [0, 1, 9, 4, 2, 3],
    ...     'col4': ['a', 'B', 'c', 'D', 'e', 'F']
    ... })
    >>> df
      col1  col2  col3 col4
    0    A     2     0    a
    1    A     1     1    B
    2    B     9     9    c
```

```
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F

Sort by col1

>>> df.sort_values(by=['col1'])
  col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D

Sort by multiple columns

>>> df.sort_values(by=['col1', 'col2'])
  col1  col2  col3 col4
1    A     1     1    B
0    A     2     0    a
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D

Sort Descending

>>> df.sort_values(by='col1', ascending=False)
  col1  col2  col3 col4
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B
3  NaN     8     4    D

Putting NAs first

>>> df.sort_values(by='col1', ascending=False, na_position='first')
  col1  col2  col3 col4
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B

Sorting with a key function

>>> df.sort_values(by='col4', key=lambda col: col.str.lower())
   col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F

Natural sort with the key argument,
using the `natsort <https://github.com/SethMMorton/natsort>` package.

>>> df = pd.DataFrame({
...    "time": ['0hr', '128hr', '72hr', '48hr', '96hr'],
...    "value": [10, 20, 30, 40, 50]
... })
>>> df
    time  value
0    0hr     10
1  128hr     20
```

```
2    72hr     30
3    48hr     40
4    96hr     50
>>> from natsort import index_natsorted
>>> df.sort_values(
...     by="time",
...     key=lambda x: np.argsort(index_natsorted(df["time"]))
... )
    time  value
0    0hr     10
3   48hr     40
2   72hr     30
4   96hr     50
1  128hr     20
```

In [30]:
```
head.to_numpy()
```

Out[30]:
```
array([[7129300520, '20141013T000000', 221900.0, 3, 1.0, 1180, 5650, 1.0,
        0, 0, 3, 7, 1180, 0, 1955, 0, 98178, 47.5112, -122.257, 1340,
        5650],
       [6414100192, '20141209T000000', 538000.0, 3, 2.25, 2570, 7242,
        2.0, 0, 0, 3, 7, 2170, 400, 1951, 1991, 98125, 47.721, -122.319,
        1690, 7639],
       [5631500400, '20150225T000000', 180000.0, 2, 1.0, 770, 10000, 1.0,
        0, 0, 3, 6, 770, 0, 1933, 0, 98028, 47.7379, -122.233, 2720,
        8062],
       [2487200875, '20141209T000000', 604000.0, 4, 3.0, 1960, 5000, 1.0,
        0, 0, 5, 7, 1050, 910, 1965, 0, 98136, 47.5208, -122.393, 1360,
        5000],
       [1954400510, '20150218T000000', 510000.0, 3, 2.0, 1680, 8080, 1.0,
        0, 0, 3, 8, 1680, 0, 1987, 0, 98074, 47.6168, -122.045, 1800,
        7503]], dtype=object)
```

In [31]:
```
help(head.to_numpy)
```

```
Help on method to_numpy in module pandas.core.frame:

to_numpy(dtype=None, copy: 'bool' = False, na_value=<object object at 0x7fc1f54b2e00
>) -> 'np.ndarray' method of pandas.core.frame.DataFrame instance
    Convert the DataFrame to a NumPy array.

    .. versionadded:: 0.24.0

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

    Parameters
    ----------
    dtype : str or numpy.dtype, optional
        The dtype to pass to :meth:`numpy.asarray`.
    copy : bool, default False
        Whether to ensure that the returned value is not a view on
        another array. Note that ``copy=False`` does not *ensure* that
        ``to_numpy()`` is no-copy. Rather, ``copy=True`` ensure that
        a copy is made, even if not strictly necessary.
    na_value : Any, optional
        The value to use for missing values. The default value depends
        on `dtype` and the dtypes of the DataFrame columns.

        .. versionadded:: 1.1.0

    Returns
    -------
    numpy.ndarray

    See Also
    --------
    Series.to_numpy : Similar method for Series.

    Examples
    --------
    >>> pd.DataFrame({"A": [1, 2], "B": [3, 4]}).to_numpy()
    array([[1, 3],
           [2, 4]])

    With heterogeneous data, the lowest common type will have to
    be used.

    >>> df = pd.DataFrame({"A": [1, 2], "B": [3.0, 4.5]})
    >>> df.to_numpy()
    array([[1. , 3. ],
           [2. , 4.5]])

    For a mix of numeric and non-numeric types, the output array will
    have object dtype.

    >>> df['C'] = pd.date_range('2000', periods=2)
    >>> df.to_numpy()
    array([[1, 3.0, Timestamp('2000-01-01 00:00:00')],
           [2, 4.5, Timestamp('2000-01-02 00:00:00')]], dtype=object)
```

Selection

Selection by label (`.loc`) or by position (`.iloc`)

First recall the basic slicing for Series

In [32]:
```
s2
```

Out[32]:
```
a    2
b    4
c    6
dtype: int64
```

In [33]:
```
s2[0:2] # by position
```

Out[33]:
```
a    2
b    4
dtype: int64
```

In [34]:
```
s2['a':'c'] # by label, the last index is INCLUDED!!!
```

Out[34]:
```
a    2
b    4
c    6
dtype: int64
```

In [35]:
```
s2.index
```

Out[35]:
```
Index(['a', 'b', 'c'], dtype='object')
```

However, confusions may occur if the "labels" are very similar to "position"

In [36]:
```
s3= pd.Series(['a','b','c','d','e'])
s3
```

Out[36]:
```
0    a
1    b
2    c
3    d
4    e
dtype: object
```

In [37]:
```
s3.index
```

Out[37]:
```
RangeIndex(start=0, stop=5, step=1)
```

In [38]:
```
s3[0:2] #slicing -- this is confusing, although it is still by position
```

Out[38]:
```
0    a
1    b
dtype: object
```

That's why pandas use `.loc` and `.iloc` to strictly distinguish by label or by position.

In [39]:
```
s3.loc[0:2] # by label
```

Out[39]:
```
0    a
1    b
2    c
dtype: object
```

In [40]:
```
s3.iloc[0:2] # by position
```

Out[40]:
```
0    a
1    b
dtype: object
```

The same applies to DataFrame.

In [41]:
```
head
```

Out[41]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	...
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	...
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	0	0	...
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	0	0	...
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	0	0	...

5 rows × 21 columns

In [42]:
```
head.iloc[:3,:2]
```

Out[42]:

	id	date
0	7129300520	20141013T000000
1	6414100192	20141209T000000
2	5631500400	20150225T000000

In [43]:
```
head.loc[:3,:'date' ]
```

Out[43]:

	id	date
0	7129300520	20141013T000000
1	6414100192	20141209T000000
2	5631500400	20150225T000000
3	2487200875	20141209T000000

Note: in the latest version of Pandas, the mixing selection .ix is ***deprecated*** *-- note this when reading the Data Science Handbook!*

In [44]:
```
help(head.loc)
```

```
Help on _LocIndexer in module pandas.core.indexing object:

class _LocIndexer(_LocationIndexer)
 |  Access a group of rows and columns by label(s) or a boolean array.
 |
 |  ``.loc[]`` is primarily label based, but may also be used with a
 |  boolean array.
 |
 |  Allowed inputs are:
 |
 |  - A single label, e.g. ``5`` or ``'a'``, (note that ``5`` is
 |    interpreted as a *label* of the index, and **never** as an
 |    integer position along the index).
 |  - A list or array of labels, e.g. ``['a', 'b', 'c']``.
 |  - A slice object with labels, e.g. ``'a':'f'``.
 |
 |    .. warning:: Note that contrary to usual python slices, **both** the
 |        start and the stop are included
 |
 |  - A boolean array of the same length as the axis being sliced,
 |    e.g. ``[True, False, True]``.
 |  - An alignable boolean Series. The index of the key will be aligned before
 |    masking.
 |  - An alignable Index. The Index of the returned selection will be the input.
 |  - A ``callable`` function with one argument (the calling Series or
 |    DataFrame) and that returns valid output for indexing (one of the above)
 |
 |  See more at :ref:`Selection by Label <indexing.label>`.
 |
 |  Raises
 |  ------
 |  KeyError
 |      If any items are not found.
 |  IndexingError
 |      If an indexed key is passed and its index is unalignable to the frame index.
 |
 |  See Also
 |  --------
 |  DataFrame.at : Access a single value for a row/column label pair.
 |  DataFrame.iloc : Access group of rows and columns by integer position(s).
 |  DataFrame.xs : Returns a cross-section (row(s) or column(s)) from the
 |      Series/DataFrame.
 |  Series.loc : Access group of values using labels.
 |
 |  Examples
 |  --------
 |  **Getting values**
 |
 |  >>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
 |  ...      index=['cobra', 'viper', 'sidewinder'],
 |  ...      columns=['max_speed', 'shield'])
 |  >>> df
 |              max_speed  shield
 |  cobra               1       2
 |  viper               4       5
 |  sidewinder          7       8
 |
 |  Single label. Note this returns the row as a Series.
 |
 |  >>> df.loc['viper']
 |  max_speed    4
 |  shield       5
 |  Name: viper, dtype: int64
 |
 |  List of labels. Note using ``[[]]`` returns a DataFrame.
 |
 |  >>> df.loc[['viper', 'sidewinder']]
 |              max_speed  shield
 |  viper               4       5
 |  sidewinder          7       8
```

```
|
|  Single label for row and column
|
|  >>> df.loc['cobra', 'shield']
|  2
|
|  Slice with labels for row and single label for column. As mentioned
|  above, note that both the start and stop of the slice are included.
|
|  >>> df.loc['cobra':'viper', 'max_speed']
|  cobra    1
|  viper    4
|  Name: max_speed, dtype: int64
|
|  Boolean list with the same length as the row axis
|
|  >>> df.loc[[False, False, True]]
|              max_speed  shield
|  sidewinder          7       8
|
|  Alignable boolean Series:
|
|  >>> df.loc[pd.Series([False, True, False],
|  ...        index=['viper', 'sidewinder', 'cobra'])]
|              max_speed  shield
|  sidewinder          7       8
|
|  Index (same behavior as ``df.reindex``)
|
|  >>> df.loc[pd.Index(["cobra", "viper"], name="foo")]
|         max_speed  shield
|  foo
|  cobra          1       2
|  viper          4       5
|
|  Conditional that returns a boolean Series
|
|  >>> df.loc[df['shield'] > 6]
|              max_speed  shield
|  sidewinder          7       8
|
|  Conditional that returns a boolean Series with column labels specified
|
|  >>> df.loc[df['shield'] > 6, ['max_speed']]
|              max_speed
|  sidewinder          7
|
|  Callable that returns a boolean Series
|
|  >>> df.loc[lambda df: df['shield'] == 8]
|              max_speed  shield
|  sidewinder          7       8
|
|  **Setting values**
|
|  Set value for all items matching the list of labels
|
|  >>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
|  >>> df
|              max_speed  shield
|  cobra               1       2
|  viper               4      50
|  sidewinder          7      50
|
|  Set value for an entire row
|
|  >>> df.loc['cobra'] = 10
|  >>> df
|              max_speed  shield
|  cobra              10      10
```

```
 |  viper                4      50
 |  sidewinder           7      50
 |
 |  Set value for an entire column
 |
 |  >>> df.loc[:, 'max_speed'] = 30
 |  >>> df
 |              max_speed  shield
 |  cobra              30      10
 |  viper              30      50
 |  sidewinder         30      50
 |
 |  Set value for rows matching callable condition
 |
 |  >>> df.loc[df['shield'] > 35] = 0
 |  >>> df
 |              max_speed  shield
 |  cobra              30      10
 |  viper               0       0
 |  sidewinder          0       0
 |
 |  **Getting values on a DataFrame with an index that has integer labels**
 |
 |  Another example using integers for the index
 |
 |  >>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
 |  ...      index=[7, 8, 9], columns=['max_speed', 'shield'])
 |  >>> df
 |     max_speed  shield
 |  7          1       2
 |  8          4       5
 |  9          7       8
 |
 |  Slice with integer labels for rows. As mentioned above, note that both
 |  the start and stop of the slice are included.
 |
 |  >>> df.loc[7:9]
 |     max_speed  shield
 |  7          1       2
 |  8          4       5
 |  9          7       8
 |
 |  **Getting values with a MultiIndex**
 |
 |  A number of examples using a DataFrame with a MultiIndex
 |
 |  >>> tuples = [
 |  ...    ('cobra', 'mark i'), ('cobra', 'mark ii'),
 |  ...    ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
 |  ...    ('viper', 'mark ii'), ('viper', 'mark iii')
 |  ... ]
 |  >>> index = pd.MultiIndex.from_tuples(tuples)
 |  >>> values = [[12, 2], [0, 4], [10, 20],
 |  ...         [1, 4], [7, 1], [16, 36]]
 |  >>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
 |  >>> df
 |                       max_speed  shield
 |  cobra      mark i           12       2
 |             mark ii           0       4
 |  sidewinder mark i           10      20
 |             mark ii           1       4
 |  viper      mark ii           7       1
 |             mark iii         16      36
 |
 |  Single label. Note this returns a DataFrame with a single index.
 |
 |  >>> df.loc['cobra']
 |          max_speed  shield
 |  mark i          12       2
 |  mark ii          0       4
```

```
 |
 |  Single index tuple. Note this returns a Series.
 |
 |  >>> df.loc[('cobra', 'mark ii')]
 |  max_speed    0
 |  shield       4
 |  Name: (cobra, mark ii), dtype: int64
 |
 |  Single label for row and column. Similar to passing in a tuple, this
 |  returns a Series.
 |
 |  >>> df.loc['cobra', 'mark i']
 |  max_speed    12
 |  shield        2
 |  Name: (cobra, mark i), dtype: int64
 |
 |  Single tuple. Note using ``[[]]`` returns a DataFrame.
 |
 |  >>> df.loc[[('cobra', 'mark ii')]]
 |                 max_speed  shield
 |  cobra mark ii          0       4
 |
 |  Single tuple for the index with a single label for the column
 |
 |  >>> df.loc[('cobra', 'mark i'), 'shield']
 |  2
 |
 |  Slice from index tuple to single label
 |
 |  >>> df.loc[('cobra', 'mark i'):'viper']
 |                       max_speed  shield
 |  cobra      mark i           12       2
 |             mark ii           0       4
 |  sidewinder mark i           10      20
 |             mark ii           1       4
 |  viper      mark ii           7       1
 |             mark iii         16      36
 |
 |  Slice from index tuple to index tuple
 |
 |  >>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
 |                      max_speed  shield
 |  cobra      mark i          12       2
 |             mark ii          0       4
 |  sidewinder mark i          10      20
 |             mark ii          1       4
 |  viper      mark ii          7       1
 |
 |  Method resolution order:
 |      _LocIndexer
 |      _LocationIndexer
 |      pandas._libs.indexing.NDFrameIndexerBase
 |      builtins.object
 |
 |  Data and other attributes defined here:
 |
 |  __annotations__ = {'_takeable': <class 'bool'>}
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from _LocationIndexer:
 |
 |  __call__(self, axis=None)
 |      Call self as a function.
 |
 |  __getitem__(self, key)
 |
 |  __setitem__(self, key, value)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from _LocationIndexer:
```

```
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from _LocationIndexer:
 |
 |  axis = None
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from pandas._libs.indexing.NDFrameIndexerBase:
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __reduce__ = __reduce_cython__(...)
 |
 |  __setstate__ = __setstate_cython__(...)
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from pandas._libs.indexing.NDFrameIndexerBase:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from pandas._libs.indexing.NDFrameIndexerBase:
 |
 |  name
 |
 |  ndim
 |
 |  obj
```

In [45]:

```
help(head.iloc)
```

```
Help on _iLocIndexer in module pandas.core.indexing object:

class _iLocIndexer(_LocationIndexer)
 |  Purely integer-location based indexing for selection by position.
 |
 |  ``.iloc[]`` is primarily integer position based (from ``0`` to
 |  ``length-1`` of the axis), but may also be used with a boolean
 |  array.
 |
 |  Allowed inputs are:
 |
 |  - An integer, e.g. ``5``.
 |  - A list or array of integers, e.g. ``[4, 3, 0]``.
 |  - A slice object with ints, e.g. ``1:7``.
 |  - A boolean array.
 |  - A ``callable`` function with one argument (the calling Series or
 |    DataFrame) and that returns valid output for indexing (one of the above).
 |    This is useful in method chains, when you don't have a reference to the
 |    calling object, but would like to base your selection on some value.
 |
 |  ``.iloc`` will raise ``IndexError`` if a requested indexer is
 |  out-of-bounds, except *slice* indexers which allow out-of-bounds
 |  indexing (this conforms with python/numpy *slice* semantics).
 |
 |  See more at :ref:`Selection by Position <indexing.integer>`.
 |
 |  See Also
 |  --------
 |  DataFrame.iat : Fast integer location scalar accessor.
 |  DataFrame.loc : Purely label-location based indexer for selection by label.
 |  Series.iloc : Purely integer-location based indexing for
 |                 selection by position.
 |
 |  Examples
 |  --------
 |  >>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
 |  ...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
 |  ...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
 |  >>> df = pd.DataFrame(mydict)
 |  >>> df
 |        a     b     c     d
 |  0     1     2     3     4
 |  1   100   200   300   400
 |  2  1000  2000  3000  4000
 |
 |  **Indexing just the rows**
 |
 |  With a scalar integer.
 |
 |  >>> type(df.iloc[0])
 |  <class 'pandas.core.series.Series'>
 |  >>> df.iloc[0]
 |  a    1
 |  b    2
 |  c    3
 |  d    4
 |  Name: 0, dtype: int64
 |
 |  With a list of integers.
 |
 |  >>> df.iloc[[0]]
 |     a  b  c  d
 |  0  1  2  3  4
 |  >>> type(df.iloc[[0]])
 |  <class 'pandas.core.frame.DataFrame'>
 |
 |  >>> df.iloc[[0, 1]]
 |       a    b    c    d
 |  0    1    2    3    4
 |  1  100  200  300  400
```

```
|
|  With a `slice` object.
|
|  >>> df.iloc[:3]
|        a     b     c     d
|  0     1     2     3     4
|  1   100   200   300   400
|  2  1000  2000  3000  4000
|
|  With a boolean mask the same length as the index.
|
|  >>> df.iloc[[True, False, True]]
|        a     b     c     d
|  0     1     2     3     4
|  2  1000  2000  3000  4000
|
|  With a callable, useful in method chains. The `x` passed
|  to the ``lambda`` is the DataFrame being sliced. This selects
|  the rows whose index label even.
|
|  >>> df.iloc[lambda x: x.index % 2 == 0]
|        a     b     c     d
|  0     1     2     3     4
|  2  1000  2000  3000  4000
|
|  **Indexing both axes**
|
|  You can mix the indexer types for the index and columns. Use ``:`` to
|  select the entire axis.
|
|  With scalar integers.
|
|  >>> df.iloc[0, 1]
|  2
|
|  With lists of integers.
|
|  >>> df.iloc[[0, 2], [1, 3]]
|        b     d
|  0     2     4
|  2  2000  4000
|
|  With `slice` objects.
|
|  >>> df.iloc[1:3, 0:3]
|        a     b     c
|  1   100   200   300
|  2  1000  2000  3000
|
|  With a boolean array whose length matches the columns.
|
|  >>> df.iloc[:, [True, False, True, False]]
|        a     c
|  0     1     3
|  1   100   300
|  2  1000  3000
|
|  With a callable function that expects the Series or DataFrame.
|
|  >>> df.iloc[:, lambda df: [0, 2]]
|        a     c
|  0     1     3
|  1   100   300
|  2  1000  3000
|
|  Method resolution order:
|      _iLocIndexer
|      _LocationIndexer
|      pandas._libs.indexing.NDFrameIndexerBase
|      builtins.object
```

```
 |
 |  Methods inherited from _LocationIndexer:
 |
 |  __call__(self, axis=None)
 |      Call self as a function.
 |
 |  __getitem__(self, key)
 |
 |  __setitem__(self, key, value)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from _LocationIndexer:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from _LocationIndexer:
 |
 |  __annotations__ = {'_valid_types': <class 'str'>}
 |
 |  axis = None
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from pandas._libs.indexing.NDFrameIndexerBase:
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __reduce__ = __reduce_cython__(...)
 |
 |  __setstate__ = __setstate_cython__(...)
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from pandas._libs.indexing.NDFrameIndexerBase:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from pandas._libs.indexing.NDFrameIndexerBase:
 |
 |  name
 |
 |  ndim
 |
 |  obj
```

In [46]:
```
head.loc[0,'price']
head.at[0,'price'] # .at can only access to one value
```

Out[46]: 221900.0

In [47]:

```
help(head.at)
```

```
Help on _AtIndexer in module pandas.core.indexing object:

class _AtIndexer(_ScalarAccessIndexer)
 |  Access a single value for a row/column label pair.
 |
 |  Similar to ``loc``, in that both provide label-based lookups. Use
 |  ``at`` if you only need to get or set a single value in a DataFrame
 |  or Series.
 |
 |  Raises
 |  ------
 |  KeyError
 |      If 'label' does not exist in DataFrame.
 |
 |  See Also
 |  --------
 |  DataFrame.iat : Access a single value for a row/column pair by integer
 |      position.
 |  DataFrame.loc : Access a group of rows and columns by label(s).
 |  Series.at : Access a single value using a label.
 |
 |  Examples
 |  --------
 |  >>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
 |  ...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
 |  >>> df
 |      A   B   C
 |  4   0   2   3
 |  5   0   4   1
 |  6  10  20  30
 |
 |  Get value at specified row/column pair
 |
 |  >>> df.at[4, 'B']
 |  2
 |
 |  Set value at specified row/column pair
 |
 |  >>> df.at[4, 'B'] = 10
 |  >>> df.at[4, 'B']
 |  10
 |
 |  Get value within a Series
 |
 |  >>> df.loc[5].at['B']
 |  4
 |
 |  Method resolution order:
 |      _AtIndexer
 |      _ScalarAccessIndexer
 |      pandas._libs.indexing.NDFrameIndexerBase
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __getitem__(self, key)
 |
 |  __setitem__(self, key, value)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from _ScalarAccessIndexer:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from pandas._libs.indexing.NDFrameIndexerBase:
```

```
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __reduce__ = __reduce_cython__(...)
 |
 |  __setstate__ = __setstate_cython__(...)
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from pandas._libs.indexing.NDFrameIndexerBase:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from pandas._libs.indexing.NDFrameIndexerBase:
 |
 |  name
 |
 |  ndim
 |
 |  obj
```

More Comments on Slicing and Indexing in DataFrame

Slicing picks rows, while indexing picks columns -- this can be confusing, and that's why `.iloc` and `.loc` are more strict.

General Rule: Direct **slicing** applies to rows and **indexing** (simple or fancy) applies to columns. If we want more flexible and convenient usage, please use `.iloc` and `.loc` .

In [48]:
```
head['date'] #same with head.date, indexing -column, no problem
```

Out[48]:
```
0    20141013T000000
1    20141209T000000
2    20150225T000000
3    20141209T000000
4    20150218T000000
Name: date, dtype: object
```

In [49]:
```
head[['date','price']] # fancy indexing -column, no problem
```

Out[49]:

	date	price
0	20141013T000000	221900.0
1	20141209T000000	538000.0
2	20150225T000000	180000.0
3	20141209T000000	604000.0
4	20150218T000000	510000.0

In [50]:
```
head[['date']] # fancy indexing -column, no problem, get the dataframe instead of ser
ies
```

Out[50]:

	date
0	20141013T000000
1	20141209T000000
2	20150225T000000
3	20141209T000000
4	20150218T000000

```
In [51]: head[0:2] #slicing -- rows
```

Out[51]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	...
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	...

2 rows × 21 columns

```
In [52]: head['date':'price'] # this is wrong -- slicing cannot be applied to rows!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-52-4e474bdfffd7> in <module>
----> 1 head['date':'price'] # this is wrong -- slicing cannot be applied to rows!

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/frame.py in __getitem__(sel
f, key)
   2997 
   2998         # Do we have a slicer (on rows)?
-> 2999         indexer = convert_to_index_sliceable(self, key)
   3000         if indexer is not None:
   3001             if isinstance(indexer, np.ndarray):

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/indexing.py in convert_to_in
dex_sliceable(obj, key)
   2205     idx = obj.index
   2206     if isinstance(key, slice):
-> 2207         return idx._convert_slice_indexer(key, kind="getitem")
   2208 
   2209     elif isinstance(key, str):

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py in _convert_
slice_indexer(self, key, kind)
   3354             """
   3355             if self.is_integer() or is_index_slice:
-> 3356                 self._validate_indexer("slice", key.start, "getitem")
   3357                 self._validate_indexer("slice", key.stop, "getitem")
   3358                 self._validate_indexer("slice", key.step, "getitem")

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py in _validate
_indexer(self, form, key, kind)
   5307             pass
   5308         else:
-> 5309             raise self._invalid_indexer(form, key)
   5310 
   5311     def _maybe_cast_slice_bound(self, label, side: str_t, kind):

TypeError: cannot do slice indexing on RangeIndex with these indexers [date] of type
str
```

In [53]:
```python
head[:,'date':'price']# this is also wrong!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-53-963ada82415c> in <module>
----> 1 head[:,'date':'price']# this is also wrong!

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/frame.py in __getitem__(sel
f, key)
   3022             if self.columns.nlevels > 1:
   3023                 return self._getitem_multilevel(key)
-> 3024             indexer = self.columns.get_loc(key)
   3025             if is_integer(indexer):
   3026                 indexer = [indexer]

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py in get_loc(s
elf, key, method, tolerance)
   3078             casted_key = self._maybe_cast_indexer(key)
   3079             try:
-> 3080                 return self._engine.get_loc(casted_key)
   3081             except KeyError as err:
   3082                 raise KeyError(key) from err

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

TypeError: '(slice(None, None, None), slice('date', 'price', None))' is an invalid k
ey
```

In [54]:
```python
head[:,['date','price']] # this is also wrong!! -- cannot do both!!!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-54-585d464c5f17> in <module>
----> 1 head[:,['date','price']] # this is also wrong!! -- cannot do both!!!

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/frame.py in __getitem__(sel
f, key)
   3022             if self.columns.nlevels > 1:
   3023                 return self._getitem_multilevel(key)
-> 3024             indexer = self.columns.get_loc(key)
   3025             if is_integer(indexer):
   3026                 indexer = [indexer]

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py in get_loc(s
elf, key, method, tolerance)
   3078             casted_key = self._maybe_cast_indexer(key)
   3079             try:
-> 3080                 return self._engine.get_loc(casted_key)
   3081             except KeyError as err:
   3082                 raise KeyError(key) from err

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

TypeError: '(slice(None, None, None), ['date', 'price'])' is an invalid key
```

In [55]:
```python
head[1:3][['date','price']] # to do slicing and indexing "simultaneously", you have t
o do them separately!
```

Out[55]:

	date	price
1	20141209T000000	538000.0
2	20150225T000000	180000.0

In [56]:
```python
head.loc[:,'date':'price'] # no problem for slicing in .loc
```

Out[56]:

	date	price
0	20141013T000000	221900.0
1	20141209T000000	538000.0
2	20150225T000000	180000.0
3	20141209T000000	604000.0
4	20150218T000000	510000.0

In [57]:
```python
head.loc[:,['date','price']] # fancy indexing is also supported in .loc
```

Out[57]:

	date	price
0	20141013T000000	221900.0
1	20141209T000000	538000.0
2	20150225T000000	180000.0
3	20141209T000000	604000.0
4	20150218T000000	510000.0

In [58]:
```python
states
```

Out[58]:

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

In [59]:
```python
states['California':'Texas']
```

Out[59]:

	population	area
California	38332521	423967
Texas	26448193	695662

In [60]:
```python
states['population']
```

Out[60]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
Name: population, dtype: int64
```

In [61]:
```
states['California':'Texas','population'] # this is wrong, cannot do both!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-61-048ac2c79b68> in <module>
----> 1 states['California':'Texas','population'] # this is wrong, cannot do both!

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/frame.py in __getitem__(sel
f, key)
   3022             if self.columns.nlevels > 1:
   3023                 return self._getitem_multilevel(key)
-> 3024             indexer = self.columns.get_loc(key)
   3025             if is_integer(indexer):
   3026                 indexer = [indexer]

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py in get_loc(s
elf, key, method, tolerance)
   3078             casted_key = self._maybe_cast_indexer(key)
   3079             try:
-> 3080                 return self._engine.get_loc(casted_key)
   3081             except KeyError as err:
   3082                 raise KeyError(key) from err

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

TypeError: '(slice('California', 'Texas', None), 'population')' is an invalid key
```

In [62]:
```
states.loc['California':'Texas','population']
```

Out[62]:
```
California    38332521
Texas         26448193
Name: population, dtype: int64
```

In [63]:
```
states.loc['California':'Texas']
```

Out[63]:

	population	area
California	38332521	423967
Texas	26448193	695662

Boolean Selection

In [ ]:
```
ind = states.area>200000
ind
```

In [ ]:
```
states[ind]
```

In [ ]:
```
states[ind,'area'] # this is wrong!
```

In [ ]:
```
states[ind]['area']
```

In [ ]:
```
states.loc[states.area>200000,'population'] # equivalently, states.loc[ind,'populatio
n']
```

In [ ]:
```
states.iloc[ind.to_numpy(),1] # in iloc, the boolen should be the Numpy array
```

In [ ]:
```
random
```

In [ ]:
```
random[random['foo']>0.6]
```

In [ ]:
```
house_price
```

Sometimes it's very useful to use the `isin` method to filter samples.

In [ ]:
```
house_price[house_price.loc[:,'bedrooms'].isin([2,4])]
```

In [ ]:
```
house_price[house_price['bedrooms'].isin([2,4])] # the same with column index
```

In [ ]:
```
house_price[(house_price['bedrooms']==2)|(house_price['bedrooms']==4)] #equivalent way
```

Basic Manipulation

- Rename

In [ ]:
```
states
```

In [ ]:
```
states_new = states.rename(columns = {"population":"Population","area":"Area"},index = {"New York":"NewYork"}) # return a new one -- if don't want to, specify inplace = True
states_new
```

In [ ]:
```
help(states.rename)
```

- Append/Drop

In [ ]:
```
states
```

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

In [ ]:
```
new_row = pd.DataFrame({'population':7614893, 'area':184827},index = ['Washington'])
new_row
```

In [ ]:
```
states_new = states.append(new_row)
states_new
```

In [ ]:
```
states_new.drop(index = "Washington",columns = "density",inplace = True)
states_new
```

- Concatenation

`pd.concat()` is a function while `.append()` is a method

In [ ]:
```
states_new1 = pd.concat([states,new_row])
states_new1
```

In [ ]:
```
states_new
```

In [ ]:
```
pd.concat([states_new,states_new1.loc[:"Illinois","density"]],axis = 1)
```

In [ ]:
```
help(pd.concat)
```

- Merge: "Concat by Value"

In [ ]:
```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
```

In [ ]:
```
df1
```

In [ ]:
```
df2
```

In [ ]:
```
pd.concat([df1,df2])
```

In [ ]:
```
pd.concat([df1,df2],axis=1)
```

In [ ]:
```
pd.merge(df1,df2)
```

In [ ]:
```
df3 = pd.merge(df1,df2,on="employee")
df3
```

In [ ]:
```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
df4
```

In [ ]:
```
pd.merge(df3,df4)
```