Section 8 Introduction to Pandas

Pandas--*Python Data Analysis Library* 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.

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

In [2]:
```
pip install pandas --upgrade
```

```
Requirement already satisfied: pandas in e:\programdata\anaconda3\lib\site-packages (1.
3.0)
Requirement already satisfied: python-dateutil>=2.7.3 in e:\programdata\anaconda3\lib\si
te-packages (from pandas) (2.8.1)
Requirement already satisfied: pytz>=2017.3 in e:\programdata\anaconda3\lib\site-package
s (from pandas) (2021.1)
Requirement already satisfied: numpy>=1.17.3 in e:\programdata\anaconda3\lib\site-packag
es (from pandas) (1.20.1)
Requirement already satisfied: six>=1.5 in e:\programdata\anaconda3\lib\site-packages (f
rom python-dateutil>=2.7.3->pandas) (1.15.0)
Note: you may need to restart the kernel to use updated packages.
```

In [2]:
```
pd.__version__
```

Out[2]:
```
'1.3.0'
```

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

```
0    2
1    4
2    6
dtype: int64
```

In [5]:
```
type(s1)
```

Out[5]:
```
pandas.core.series.Series
```

In [6]:
```
s1.index
```

Out[6]:
```
RangeIndex(start=0, stop=3, step=1)
```

In [7]:
```
s2 = pd.Series([2, 4, 6],index = ['a','b','c'])
```

In [8]:
```
s2
```

Out[8]:
```
a    2
b    4
c    6
dtype: int64
```

In [11]:
```
s2_num = s2.values # change to Numpy -- can be view instead of copy if the elements are
s2_num
```

Out[11]:
```
array([2, 4, 6], dtype=int64)
```

In [12]:
```
s2.index
```

Out[12]:
```
Index(['a', 'b', 'c'], dtype='object')
```

In [13]:
```
np.shares_memory(s2_num,s2)
```

Out[13]:
```
True
```

In [14]:
```
s2_num_copy = s2.to_numpy(copy = True) # more recommended in new version of Pandas -- c
np.shares_memory(s2_num_copy,s2)
```

Out[14]:
```
False
```

Selection by position -- similar to Numpy array!

In [17]:
```
s2[0:2]
```

Out[17]:
```
a    2
b    4
dtype: int64
```

Selection by index (label)

In [18]:
```
s2['a']
```

Out[18]:
```
2
```

In [20]:
```
s2[['a','c']]
```

Out[20]:
```
a    2
c    6
dtype: int64
```

`Series` and Python Dictionary

In [22]:
```
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
print(population_dict)
print(population)
```

```
{'California': 38332521, 'Texas': 26448193, 'New York': 19651127, 'Florida': 19552860,
'Illinois': 12882135}
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

In [23]:
```
population_dict['Texas'] # key and value
```

Out[23]:
```
26448193
```

In [24]:
```
population['Texas']
```

Out[24]:
```
26448193
```

In [25]:
```
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995} #Note: units are km^2
area = pd.Series(area_dict)
area
```

Out[25]:
```
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 [26]:
```
states = pd.DataFrame({'Population': population,
                       'Area': area}) # variable names
states
```

Out[26]:

	Population	Area
California	38332521	423967

	Population	Area
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

In [27]:
```
type(states)
```

Out[27]:
```
pandas.core.frame.DataFrame
```

In [28]:
```
states.index
```

Out[28]:
```
Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='object')
```

In [40]:
```
states.columns
```

Out[40]:
```
Index(['Population', 'Area'], dtype='object')
```

In [41]:
```
states['Area']
```

Out[41]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: Area, dtype: int64
```

In [42]:
```
states.Area
```

Out[42]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: Area, dtype: int64
```

In [29]:
```
type(states['Area'])
```

Out[29]:
```
pandas.core.series.Series
```

In [32]:
```
random = pd.DataFrame(np.random.rand(3, 2),columns=['ipsum', 'lorem'],index=['A', 'B',
random
```

Out[32]:

	ipsum	lorem
A	0.883742	0.065904
B	0.524140	0.415648

	ipsum	lorem
C	0.901455	0.490729

In [33]:
```
random.T
```

Out[33]:

	A	B	C
ipsum	0.883742	0.524140	0.901455
lorem	0.065904	0.415648	0.490729

Creating DataFrame from Files

In [35]:
```
house_price = pd.read_csv('kc_house_data.csv')
house_price
```

Out[35]:

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

21613 rows × 21 columns

In [36]:
```
house_price.shape # dimension of the data
```

Out[36]:
```
(21613, 21)
```

In [37]:
```
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 [47]:
```
house_price.head(3) # show the head lines
```

Out[47]:

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

3 rows × 21 columns

In [44]:
```
house_price.sample(5) # show the random samples
```

Out[44]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wat
17481	4140900050	20150126T000000	440000.0	4	1.75	2180	10200	1.0	
4047	4167300300	20140813T000000	310000.0	4	1.75	1880	12150	1.0	
7016	1446400670	20140731T000000	199950.0	3	1.50	1510	6600	1.0	
17733	1245500286	20140523T000000	498000.0	2	2.00	1140	8282	1.0	
20157	1102000514	20141022T000000	970000.0	5	3.50	3400	9804	2.0	

5 rows × 21 columns

In [45]:
```
house_price.describe() # descriptive statistics
```

Out[45]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floc

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floc
count	2.161300e+04	2.161300e+04	21613.000000	21613.000000	21613.000000	2.161300e+04	21613.0000
mean	4.580302e+09	5.401822e+05	3.370842	2.114757	2079.899736	1.510697e+04	1.4943
std	2.876566e+09	3.673622e+05	0.930062	0.770163	918.440897	4.142051e+04	0.5399
min	1.000102e+06	7.500000e+04	0.000000	0.000000	290.000000	5.200000e+02	1.0000
25%	2.123049e+09	3.219500e+05	3.000000	1.750000	1427.000000	5.040000e+03	1.0000
50%	3.904930e+09	4.500000e+05	3.000000	2.250000	1910.000000	7.618000e+03	1.5000
75%	7.308900e+09	6.450000e+05	4.000000	2.500000	2550.000000	1.068800e+04	2.0000
max	9.900000e+09	7.700000e+06	33.000000	8.000000	13540.000000	1.651359e+06	3.5000

In [46]:
```
help(house_price.head)
```

```
Help on method head in module pandas.core.generic:

head(n: 'int' = 5) -> 'FrameOrSeries' method of pandas.core.frame.DataFrame instance
    Return the first `n` rows.

    This function returns the first `n` rows for the object based
    on position. It is useful for quickly testing if your object
    has the right type of data in it.

    For negative values of `n`, this function returns all rows except
    the last `n` rows, equivalent to ``df[:-n]``.

    Parameters
    ----------
    n : int, default 5
        Number of rows to select.

    Returns
    -------
    same type as caller
        The first `n` rows of the caller object.

    See Also
    --------
    DataFrame.tail: Returns the last `n` rows.

    Examples
    --------
    >>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
    ...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
    >>> df
          animal
    0  alligator
    1        bee
    2     falcon
    3       lion
    4     monkey
    5     parrot
    6      shark
    7      whale
    8      zebra
```

```
Viewing the first 5 lines

>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey

Viewing the first `n` lines (three in this case)

>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon

For negative values of `n`

>>> df.head(-3)
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
```

In [49]:
```
head = house_price.head(10)
head.to_csv('head.csv')
```

In [50]:
```
head
```

Out[50]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfr
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	
5	7237550310	20140512T000000	1230000.0	4	4.50	5420	101930	1.0	
6	1321400060	20140627T000000	257500.0	3	2.25	1715	6819	2.0	
7	2008000270	20150115T000000	291850.0	3	1.50	1060	9711	1.0	
8	2414600126	20150415T000000	229500.0	3	1.00	1780	7470	1.0	
9	3793500160	20150312T000000	323000.0	3	2.50	1890	6560	2.0	

10 rows × 21 columns

In [53]:
```
head.sort_values(by='price', ascending=False)
```

Out[53]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfr
5	7237550310	20140512T000000	1230000.0	4	4.50	5420	101930	1.0	
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	
9	3793500160	20150312T000000	323000.0	3	2.50	1890	6560	2.0	
7	2008000270	20150115T000000	291850.0	3	1.50	1060	9711	1.0	
6	1321400060	20140627T000000	257500.0	3	2.25	1715	6819	2.0	
8	2414600126	20150415T000000	229500.0	3	1.00	1780	7470	1.0	
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	

10 rows × 21 columns

In [52]:

```
help(head.sort_values)
```

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

sort_values(by, axis: 'Axis' = 0, ascending=True, inplace: 'bool' = False, kind: 'str' =
'quicksort', na_position: 'str' = 'last', ignore_index: 'bool' = False, key: 'ValueKeyFu
nc' = None) method of pandas.core.frame.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', 'stable'}, default 'quicksort'
         Choice of sorting algorithm. See also :func:`numpy.sort` for more
         information. `mergesort` and `stable` are the only stable algorithms. 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 [54]:

```
head.to_numpy()
```

Out[54]:

```
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],
       [7237550310, '20140512T000000', 1230000.0, 4, 4.5, 5420, 101930,
        1.0, 0, 0, 3, 11, 3890, 1530, 2001, 0, 98053, 47.6561, -122.005,
        4760, 101930],
       [1321400060, '20140627T000000', 257500.0, 3, 2.25, 1715, 6819,
        2.0, 0, 0, 3, 7, 1715, 0, 1995, 0, 98003, 47.3097, -122.327,
        2238, 6819],
       [2008000270, '20150115T000000', 291850.0, 3, 1.5, 1060, 9711, 1.0,
        0, 0, 3, 7, 1060, 0, 1963, 0, 98198, 47.4095, -122.315, 1650,
        9711],
       [2414600126, '20150415T000000', 229500.0, 3, 1.0, 1780, 7470, 1.0,
        0, 0, 3, 7, 1050, 730, 1960, 0, 98146, 47.5123, -122.337, 1780,
        8113],
       [3793500160, '20150312T000000', 323000.0, 3, 2.5, 1890, 6560, 2.0,
        0, 0, 3, 7, 1890, 0, 2003, 0, 98038, 47.3684, -122.031, 2390,
        7570]], dtype=object)
```

In [55]:

```
help(head.to_numpy)
```

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

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

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

    Parameters
    ----------
    dtype : str or numpy.dtype, optional
        The dtype to pass to :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 [56]:
```
s2
```

Out[56]:
```
a    2
b    4
c    6
dtype: int64
```

In [57]:
```
s2[0:2] # by position, last index not included
```

Out[57]:
```
a    2
b    4
dtype: int64
```

In [58]:
```
s2['a':'c'] # by label, the last index is INCLUDED!!!
```

Out[58]:
```
a    2
b    4
c    6
dtype: int64
```

In [59]:
```
s2.index
```

Out[59]:
```
Index(['a', 'b', 'c'], dtype='object')
```

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

In [60]:
```
s3= pd.Series(['a','b','c','d','e'])
s3
```

Out[60]:
```
0    a
1    b
2    c
3    d
4    e
dtype: object
```

In [66]:
```
s3.index
```

Out[66]:
```
RangeIndex(start=0, stop=5, step=1)
```

In [67]:
```
s3[0:2] #slicing -- this is confusing, although it is still by position
```

Out[67]:
```
0    a
1    b
dtype: object
```

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

In [68]:
```
s3.loc[0:2] # by label
```

Out[68]:
```
0    a
1    b
2    c
dtype: object
```

In [66]:
```
s3.iloc[0:2] # by position.
```

Out[66]:
```
0    a
1    b
dtype: object
```

The same applies to DataFrame.

In [69]:
```
head
```

Out[69]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfr
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	
5	7237550310	20140512T000000	1230000.0	4	4.50	5420	101930	1.0	
6	1321400060	20140627T000000	257500.0	3	2.25	1715	6819	2.0	
7	2008000270	20150115T000000	291850.0	3	1.50	1060	9711	1.0	
8	2414600126	20150415T000000	229500.0	3	1.00	1780	7470	1.0	
9	3793500160	20150312T000000	323000.0	3	2.50	1890	6560	2.0	

10 rows × 21 columns

In [72]:
```
head.iloc[:3,:2] #index loc, by position
```

Out[72]:

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

In [97]:
```
head.loc[:3,'date':'floors' ] #Loc or label loc, by label
```

Out[97]:

	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors
0	20141013T000000	221900.0	3	1.00	1180	5650	1.0
1	20141209T000000	538000.0	3	2.25	2570	7242	2.0
2	20150225T000000	180000.0	2	1.00	770	10000	1.0
3	20141209T000000	604000.0	4	3.00	1960	5000	1.0

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

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

In [ ]:
```
help(head.iloc)
```

In [81]:
```
head.loc[0,'price']
```

Out[81]:
```
221900.0
```

In [82]:
```
head.at[0,'price'] # .at can only access to one value
```

Out[82]:
```
221900.0
```

In [83]:
```
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 [84]:
```
head['date'] #same with head.date, indexing -column, no problem
```

Out[84]:
```
0    20141013T000000
1    20141209T000000
2    20150225T000000
3    20141209T000000
4    20150218T000000
5    20140512T000000
6    20140627T000000
7    20150115T000000
8    20150415T000000
9    20150312T000000
Name: date, dtype: object
```

In [88]:
```
head[['date','price']] # fancy indexing -column, no problem
```

Out[88]:

	date	price
0	20141013T000000	221900.0
1	20141209T000000	538000.0
2	20150225T000000	180000.0
3	20141209T000000	604000.0
4	20150218T000000	510000.0
5	20140512T000000	1230000.0
6	20140627T000000	257500.0
7	20150115T000000	291850.0

	date	price
8	20150415T000000	229500.0
9	20150312T000000	323000.0

In [89]:
```
head[['date']] # fancy indexing -column, no problem, get the dataframe instead of serie
```

Out[89]:

	date
0	20141013T000000
1	20141209T000000
2	20150225T000000
3	20141209T000000
4	20150218T000000
5	20140512T000000
6	20140627T000000
7	20150115T000000
8	20150415T000000
9	20150312T000000

In [90]:
```
head[0:2] #slicing -- rows
```

Out[90]:

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

2 rows × 21 columns

In [92]:
```
head['date':'price'] # this is wrong -- this slicing cannot be applied to the rows!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-92-b66e6d705542> in <module>
----> 1 head['date':'price'] # this is wrong -- this slicing cannot be applied to the r
ows!

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\frame.py in __getitem__(self, ke
y)
   3428
   3429         # Do we have a slicer (on rows)?
-> 3430         indexer = convert_to_index_sliceable(self, key)
   3431         if indexer is not None:
   3432             if isinstance(indexer, np.ndarray):

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexing.py in convert_to_index_s
```

```
liceable(obj, key)
   2327     idx = obj.index
   2328     if isinstance(key, slice):
-> 2329         return idx._convert_slice_indexer(key, kind="getitem")
   2330
   2331     elif isinstance(key, str):

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\numeric.py in _convert_sl
ice_indexer(self, key, kind)
    242             return self.slice_indexer(key.start, key.stop, key.step, kind=kind)
    243
--> 244         return super()._convert_slice_indexer(key, kind=kind)
    245
    246     @doc(Index._maybe_cast_slice_bound)

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in _convert_slice
_indexer(self, key, kind)
   3717             """
   3718             if self.is_integer() or is_index_slice:
-> 3719                 self._validate_indexer("slice", key.start, "getitem")
   3720                 self._validate_indexer("slice", key.stop, "getitem")
   3721                 self._validate_indexer("slice", key.step, "getitem")

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in _validate_inde
xer(self, form, key, kind)
   5718
   5719         if key is not None and not is_integer(key):
-> 5720             raise self._invalid_indexer(form, key)
   5721
   5722     def _maybe_cast_slice_bound(self, label, side: str_t, kind=no_default):

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

In [94]:

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-94-963ada82415c> in <module>
----> 1 head[:,'date':'price']# this is also wrong!

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\frame.py in __getitem__(self, ke
y)
   3453             if self.columns.nlevels > 1:
   3454                 return self._getitem_multilevel(key)
-> 3455             indexer = self.columns.get_loc(key)
   3456             if is_integer(indexer):
   3457                 indexer = [indexer]

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_loc(self,
 key, method, tolerance)
   3359             casted_key = self._maybe_cast_indexer(key)
   3360             try:
-> 3361                 return self._engine.get_loc(casted_key)
   3362             except KeyError as err:
   3363                 raise KeyError(key) from err

E:\ProgramData\Anaconda3\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.
IndexEngine.get_loc()

E:\ProgramData\Anaconda3\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.
IndexEngine.get_loc()

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

In [96]:
```
head[:,['date','price']] # this is also wrong!! -- cannot do both!!!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-96-585d464c5f17> in <module>
----> 1 head[:,['date','price']] # this is also wrong!! -- cannot do both!!!

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\frame.py in __getitem__(self, ke
y)
   3453             if self.columns.nlevels > 1:
   3454                 return self._getitem_multilevel(key)
-> 3455             indexer = self.columns.get_loc(key)
   3456             if is_integer(indexer):
   3457                 indexer = [indexer]

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_loc(self,
 key, method, tolerance)
   3359             casted_key = self._maybe_cast_indexer(key)
   3360             try:
-> 3361                 return self._engine.get_loc(casted_key)
   3362             except KeyError as err:
   3363                 raise KeyError(key) from err

E:\ProgramData\Anaconda3\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.
IndexEngine.get_loc()

E:\ProgramData\Anaconda3\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.
IndexEngine.get_loc()

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

In [101…
```
small = head[1:3]
small
```

Out[101…

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfro
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	

2 rows × 21 columns

In [102…
```
small[['date','price']]
```

Out[102…

	date	price
1	20141209T000000	538000.0
2	20150225T000000	180000.0

In [98]:
```
head[1:3][['date','price']] # to do slicing and indexing "simultaneously", you have to
```

Out[98]:

	date	price
1	20141209T000000	538000.0

	date	price
2	20150225T000000	180000.0

In [105…
```
head.loc[1:2,'date':'price'] # no problem for slicing in .loc
```

Out[105…

	date	price
1	20141209T000000	538000.0
2	20150225T000000	180000.0

In [107…
```
head.loc[2,['date','bedrooms']] # fancy indexing is also supported in .loc
```

Out[107…
```
date        20150225T000000
bedrooms                  2
Name: 2, dtype: object
```

In [108…
```
states
```

Out[108…

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

In [109…
```
states.loc[:'New York', ['Area']]
```

Out[109…

	Area
California	423967
Texas	695662
New York	141297

In [111…
```
states['California':'Texas']
```

Out[111…

	Population	Area
California	38332521	423967
Texas	26448193	695662

In [110…
```
states['Population']
```

Out[110…
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
Name: Population, dtype: int64
```

In [113…
```
states['California':'Texas','Population'] # this is wrong, cannot do both! Need .loc or
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-113-48433d106310> in <module>
----> 1 states['California':'Texas','Population'] # this is wrong, cannot do both! Need
.loc or .iloc

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\frame.py in __getitem__(self, ke
y)
   3453             if self.columns.nlevels > 1:
   3454                 return self._getitem_multilevel(key)
-> 3455             indexer = self.columns.get_loc(key)
   3456             if is_integer(indexer):
   3457                 indexer = [indexer]

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_loc(self,
 key, method, tolerance)
   3359             casted_key = self._maybe_cast_indexer(key)
   3360             try:
-> 3361                 return self._engine.get_loc(casted_key)
   3362             except KeyError as err:
   3363                 raise KeyError(key) from err

E:\ProgramData\Anaconda3\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.
IndexEngine.get_loc()

E:\ProgramData\Anaconda3\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.
IndexEngine.get_loc()

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

In [114…
```
states.loc['California':'Texas','Population']
```

Out[114…
```
California    38332521
Texas         26448193
Name: Population, dtype: int64
```

In [115…
```
states.loc['California':'Texas']
```

Out[115…

	Population	Area
California	38332521	423967
Texas	26448193	695662

Boolean Selection

In [116…
```
ind = states.Area>200000 #states with more than 200,000 km^2 area
ind
```

Out[116…
```
California     True
Texas          True
New York      False
Florida       False
Illinois      False
Name: Area, dtype: bool
```

In [117…
```
states[ind]
```

Out[117…

	Population	Area
California	38332521	423967
Texas	26448193	695662

In [118…
```
states[ind,'area'] # this is wrong! Cannot do double indexing on just states.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-118-f4e67fb7ef1a> in <module>
----> 1 states[ind,'area'] # this is wrong! Cannot do double indexing on just states.

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\frame.py in __getitem__(self, ke
y)
   3453             if self.columns.nlevels > 1:
   3454                 return self._getitem_multilevel(key)
-> 3455             indexer = self.columns.get_loc(key)
   3456             if is_integer(indexer):
   3457                 indexer = [indexer]

E:\ProgramData\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_loc(self,
 key, method, tolerance)
   3359             casted_key = self._maybe_cast_indexer(key)
   3360             try:
-> 3361                 return self._engine.get_loc(casted_key)
   3362             except KeyError as err:
   3363                 raise KeyError(key) from err

E:\ProgramData\Anaconda3\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.
IndexEngine.get_loc()

E:\ProgramData\Anaconda3\lib\site-packages\pandas\_libs\index.pyx in pandas._libs.index.
IndexEngine.get_loc()

TypeError: '(California     True
Texas          True
New York      False
Florida       False
Illinois      False
Name: Area, dtype: bool, 'area')' is an invalid key
```

In [120…
```
states[ind]['Area']
```

Out[120…
```
California    423967
Texas         695662
Name: Area, dtype: int64
```

In [123…
```
states.loc[states.Area>200000,'Population'] # equivalently, states.loc[ind,'Population'
```

Out[123…
```
California    38332521
Texas         26448193
Name: Population, dtype: int64
```

In [124…
```
states.loc[ind,'Population']
```

Out[124…
```
California    38332521
Texas         26448193
Name: Population, dtype: int64
```

In [125…
```
states.iloc[ind.to_numpy(),1] # in iloc, the boolean should be the Numpy array
```

Out[125…
```
California    423967
Texas         695662
Name: Area, dtype: int64
```

In [128…
```
random
```

Out[128…

	ipsum	lorem
A	0.883742	0.065904
B	0.524140	0.415648
C	0.901455	0.490729

In [131…
```
random[random['ipsum']<0.6]
```

Out[131…

	ipsum	lorem
B	0.52414	0.415648

In [130…
```
random[random.ipsum<0.6]
```

Out[130…

	ipsum	lorem
B	0.52414	0.415648

In [132…
```
house_price
```

Out[132…

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

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wat
21608	263000018	20140521T000000	360000.0	3	2.50	1530	1131	3.0	
21609	6600060120	20150223T000000	400000.0	4	2.50	2310	5813	2.0	
21610	1523300141	20140623T000000	402101.0	2	0.75	1020	1350	2.0	
21611	291310100	20150116T000000	400000.0	3	2.50	1600	2388	2.0	
21612	1523300157	20141015T000000	325000.0	2	0.75	1020	1076	2.0	

21613 rows × 21 columns

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

In [136…
```
house_price[house_price.loc[:,'bedrooms'].isin([2,4])]   #either 2 bedrooms or 4 bedroo
```

Out[136…

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	
5	7237550310	20140512T000000	1230000.0	4	4.50	5420	101930	1.0	
11	9212900260	20140527T000000	468000.0	2	1.00	1160	6000	1.0	
15	9297300055	20150124T000000	650000.0	4	3.00	2950	5000	2.0	
...	...	...	...	...	...	...	...	...	...
21605	3448900210	20141014T000000	610685.0	4	2.50	2520	6023	2.0	
21606	7936000429	20150326T000000	1010000.0	4	3.50	3510	7200	2.0	
21609	6600060120	20150223T000000	400000.0	4	2.50	2310	5813	2.0	
21610	1523300141	20140623T000000	402101.0	2	0.75	1020	1350	2.0	
21612	1523300157	20141015T000000	325000.0	2	0.75	1020	1076	2.0	

9642 rows × 21 columns

In [137…
```
house_price.loc[:,'bedrooms']
```

Out[137…
```
0        3
1        3
2        2
3        4
4        3
       ..
21608    3
21609    4
21610    2
21611    3
21612    2
Name: bedrooms, Length: 21613, dtype: int64
```

In [138…
```
house_price.loc[:,'bedrooms'].isin([2,4])
```

Out[138…
```
0        False
1        False
2         True
3         True
4        False
         ...
21608    False
21609     True
21610     True
21611    False
21612     True
Name: bedrooms, Length: 21613, dtype: bool
```

In [140…
```
house_price[house_price['bedrooms'].isin([2,4])] # the same with column index
```

Out[140…

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	
5	7237550310	20140512T000000	1230000.0	4	4.50	5420	101930	1.0	
11	9212900260	20140527T000000	468000.0	2	1.00	1160	6000	1.0	
15	9297300055	20150124T000000	650000.0	4	3.00	2950	5000	2.0	
...	...	...	...	...	...	...	...	...	...
21605	3448900210	20141014T000000	610685.0	4	2.50	2520	6023	2.0	
21606	7936000429	20150326T000000	1010000.0	4	3.50	3510	7200	2.0	
21609	6600060120	20150223T000000	400000.0	4	2.50	2310	5813	2.0	
21610	1523300141	20140623T000000	402101.0	2	0.75	1020	1350	2.0	
21612	1523300157	20141015T000000	325000.0	2	0.75	1020	1076	2.0	

9642 rows × 21 columns

In [141…
```
house_price[(house_price['bedrooms']==2)|(house_price['bedrooms']==4)] #equivalent way,
```

Out[141…

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	
5	7237550310	20140512T000000	1230000.0	4	4.50	5420	101930	1.0	
11	9212900260	20140527T000000	468000.0	2	1.00	1160	6000	1.0	
15	9297300055	20150124T000000	650000.0	4	3.00	2950	5000	2.0	
...	...	...	...	...	...	...	...	...	...
21605	3448900210	20141014T000000	610685.0	4	2.50	2520	6023	2.0	

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
21606	7936000429	20150326T000000	1010000.0	4	3.50	3510	7200	2.0	
21609	6600060120	20150223T000000	400000.0	4	2.50	2310	5813	2.0	
21610	1523300141	20140623T000000	402101.0	2	0.75	1020	1350	2.0	
21612	1523300157	20141015T000000	325000.0	2	0.75	1020	1076	2.0	

9642 rows × 21 columns

Basic Manipulation

- Rename

In [142...
```
states
```

Out[142...

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

In [143...
```
states_new = states.rename(columns = {"Population":"p0pulation","Area":"area..."},index
states_new
```

Out[143...

	p0pulation	area...
California	38332521	423967
Texas	26448193	695662
NewYork	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

In [144...
```
help(states.rename)
```

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

rename(mapper=None, index=None, columns=None, axis=None, copy=True, inplace=False, level
=None, errors='ignore') method of pandas.core.frame.DataFrame instance
    Alter axes labels.

    Function / dict values must be unique (1-to-1). Labels not contained in
    a dict / Series will be left as-is. Extra labels listed don't throw an
```

```
error.

See the :ref:`user guide <basics.rename>` for more.

Parameters
----------
mapper : dict-like or function
    Dict-like or function transformations to apply to
    that axis' values. Use either ``mapper`` and ``axis`` to
    specify the axis to target with ``mapper``, or ``index`` and
    ``columns``.
index : dict-like or function
    Alternative to specifying axis (``mapper, axis=0``
    is equivalent to ``index=mapper``).
columns : dict-like or function
    Alternative to specifying axis (``mapper, axis=1``
    is equivalent to ``columns=mapper``).
axis : {0 or 'index', 1 or 'columns'}, default 0
    Axis to target with ``mapper``. Can be either the axis name
    ('index', 'columns') or number (0, 1). The default is 'index'.
copy : bool, default True
    Also copy underlying data.
inplace : bool, default False
    Whether to return a new DataFrame. If True then value of copy is
    ignored.
level : int or level name, default None
    In case of a MultiIndex, only rename labels in the specified
    level.
errors : {'ignore', 'raise'}, default 'ignore'
    If 'raise', raise a `KeyError` when a dict-like `mapper`, `index`,
    or `columns` contains labels that are not present in the Index
    being transformed.
    If 'ignore', existing keys will be renamed and extra keys will be
    ignored.

Returns
-------
DataFrame or None
    DataFrame with the renamed axis labels or None if ``inplace=True``.

Raises
------
KeyError
    If any of the labels is not found in the selected axis and
    "errors='raise'".

See Also
--------
DataFrame.rename_axis : Set the name of the axis.

Examples
--------
``DataFrame.rename`` supports two calling conventions

* ``(index=index_mapper, columns=columns_mapper, ...)``
* ``(mapper, axis={'index', 'columns'}, ...)``

We *highly* recommend using keyword arguments to clarify your
intent.

Rename columns using a mapping:

>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(columns={"A": "a", "B": "c"})
   a  c
```

```
0  1  4
1  2  5
2  3  6

Rename index using a mapping:

>>> df.rename(index={0: "x", 1: "y", 2: "z"})
   A  B
x  1  4
y  2  5
z  3  6

Cast index labels to a different type:

>>> df.index
RangeIndex(start=0, stop=3, step=1)
>>> df.rename(index=str).index
Index(['0', '1', '2'], dtype='object')

>>> df.rename(columns={"A": "a", "B": "b", "C": "c"}, errors="raise")
Traceback (most recent call last):
KeyError: ['C'] not found in axis

Using axis-style parameters:

>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6

>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

- Append/Drop

In [145…

```
states
```

Out[145…

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

In [154…

```
states['density'] = states['Population']/states['Area'] # add new column
states
```

Out[154…

	Population	Area	density
California	38332521	423967	90.413926

	Population	Area	density
Texas	26448193	695662	38.018740
New York	19651127	141297	139.076746
Florida	19552860	170312	114.806121
Illinois	12882135	149995	85.883763

In [155...
```
new_row = pd.DataFrame({'Population':7614893, 'Area':184827},index = ['Washington'])
new_row
```

Out[155...

	Population	Area
Washington	7614893	184827

In [156...
```
states_new = states.append(new_row)
states_new
```

Out[156...

	Population	Area	density
California	38332521	423967	90.413926
Texas	26448193	695662	38.018740
New York	19651127	141297	139.076746
Florida	19552860	170312	114.806121
Illinois	12882135	149995	85.883763
Washington	7614893	184827	NaN

In [157...
```
states_new_2 = states_new.drop(index = "Washington",columns = "density")
states_new.drop(index = "Washington",columns = "density",inplace = True)
states_new
```

Out[157...

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

In [158...
```
states_new_2
```

Out[158...

	Population	Area
California	38332521	423967

	Population	Area
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

- Concatenation

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

In [159…
```
states_new1 = pd.concat([states,new_row])
states_new1
```

Out[159…

	Population	Area	density
California	38332521	423967	90.413926
Texas	26448193	695662	38.018740
New York	19651127	141297	139.076746
Florida	19552860	170312	114.806121
Illinois	12882135	149995	85.883763
Washington	7614893	184827	NaN

In [160…
```
states_new #the pd.concat() function does not affect the original states_new
```

Out[160…

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

In [163…
```
pd.concat([states_new,states_new1.loc[:"Illinois","density"]],axis = 1) #concatenates u
#What does axis = 0 do?
```

Out[163…

	Population	Area	density
California	38332521	423967	90.413926
Texas	26448193	695662	38.018740
New York	19651127	141297	139.076746
Florida	19552860	170312	114.806121

	Population	Area	density
Illinois	12882135	149995	85.883763

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

- Merge: "Concat by Value"

In [165…
```
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 [166…
```
df1
```

Out[166…

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

In [167…
```
df2
```

Out[167…

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

In [168…
```
pd.concat([df1,df2])
```

Out[168…

	employee	group	hire_date
0	Bob	Accounting	NaN
1	Jake	Engineering	NaN
2	Lisa	Engineering	NaN
3	Sue	HR	NaN
0	Lisa	NaN	2004.0
1	Bob	NaN	2008.0
2	Jake	NaN	2012.0

	employee	group	hire_date
3	Sue	NaN	2014.0

In [169…
```
pd.concat([df1,df2],axis=1)
```

Out[169…

	employee	group	employee	hire_date
0	Bob	Accounting	Lisa	2004
1	Jake	Engineering	Bob	2008
2	Lisa	Engineering	Jake	2012
3	Sue	HR	Sue	2014

In [170…
```
pd.merge(df1,df2)
```

Out[170…

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

In [171…
```
df3 = pd.merge(df1,df2,on="employee")
df3
```

Out[171…

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

In [172…
```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
df4
```

Out[172…

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

In [140…
```
pd.merge(df3,df4)
```

Out[140…

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve