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

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

Out[2]: '1.2.4'

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])
```

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

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

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

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

In [15]:
```
s2 = pd.Series([2, 4, 6],index = ['a','b','c'])
```

In [16]:
```
s2
```

Out[16]:
```
a    2
```

```
b    4
c    6
dtype: int64
```

In [9]:
```
s2_num = s2.values # change to Numpy -- can be view instead of copy if the elements are
s2_num
```

Out[9]:
```
array([2, 4, 6])
```

In [10]:
```
np.shares_memory(s2_num,s2)
```

Out[10]:
```
True
```

In [11]:
```
s2_num_copy = s2.to_numpy(copy = True) # more recommended in new version of Pandas -- c
np.shares_memory(s2_num_copy,s2)
```

Out[11]:
```
False
```

Selection by position -- similar to Numpy array!

In [12]:
```
s2[0:2]
```

Out[12]:
```
a    2
b    4
dtype: int64
```

Selection by index (label)

In [14]:
```
s2['a']
```

Out[14]:
```
2
```

In [13]:
```
s2[['a','c']]
```

Out[13]:
```
a    2
c    6
dtype: int64
```

`Series` and Python Dictionary

In [49]:
```
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[49]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
```

```
Illinois      12882135
dtype: int64
```

In [16]:
```
population_dict['Texas'] # key and value
```

Out[16]:
```
26448193
```

In [18]:
```
population['Texas']
```

Out[18]:
```
26448193
```

In [50]:
```
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
area
```

Out[50]:
```
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 [51]:
```
states = pd.DataFrame({'Population': population,
                       'Area': area}) # variable names
states
```

Out[51]:

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

In [21]:
```
type(states)
```

Out[21]:
```
pandas.core.frame.DataFrame
```

In [22]:
```
states.index
```

Out[22]:
```
Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='object')
```

In [23]:
```
states.columns
```

Out[23]:
```
Index(['Population', 'Area'], dtype='object')
```

In [25]:
```
states['Area']
```

Out[25]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: Area, dtype: int64
```

In [26]:
```
states.Area
```

Out[26]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: Area, dtype: int64
```

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

Out[27]:
```
pandas.core.series.Series
```

In [28]:
```
random = pd.DataFrame(np.random.rand(3, 2),columns=['foo', 'bar'],index=['a', 'b', 'c']
random
```

Out[28]:

	foo	bar
a	0.654325	0.030998
b	0.423910	0.856790
c	0.058505	0.190484

In [29]:
```
random.T
```

Out[29]:

	a	b	c
foo	0.654325	0.42391	0.058505
bar	0.030998	0.85679	0.190484

Creating DataFrame from Files

In [2]:
```
house_price = pd.read_csv('kc_house_data.csv')
house_price
```

Out[2]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wat
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wat
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 [3]:
```
house_price.shape # dimension of the data
```

Out[3]:
```
(21613, 21)
```

In [4]:
```
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 [5]:

```
house_price.head(3) # show the head lines
```

Out[5]:

	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 [6]:
```
house_price.sample(5) # show the random samples
```

Out[6]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
12114	1823069046	20150420T000000	250000.0	3	1.50	2390	23522	1.0	
20703	5167000140	20140711T000000	1480000.0	3	3.25	3700	2264	2.0	
8076	3390600025	20140529T000000	450000.0	4	2.00	2240	7725	1.0	
9256	8562750220	20141120T000000	811500.0	5	4.25	3970	4500	2.0	
4842	6649300190	20140903T000000	407500.0	5	2.00	2740	8230	1.5	

5 rows × 21 columns

In [7]:
```
house_price.describe() # descriptive statistics
```

Out[7]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floo
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 [8]:
```
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 [9]:
```
head = house_price.head()
head.to_csv('head.csv')
```

In [10]:
```
head
```

Out[10]:

	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	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	

5 rows × 21 columns

In [11]:
```
head.sort_values(by='price')
```

Out[11]:

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

5 rows × 21 columns

In [12]:
```
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='la
st', ignore_index=False, key: 'ValueKeyFunc' = None) method of pandas.core.frame.DataFra
me 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 [13]:
```
head.to_numpy()
```

Out[13]:
```
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 [14]:
```
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 0x7fd47329ee00>) -
> '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 [17]:
```
s2
```

Out[17]:
```
a    2
b    4
c    6
dtype: int64
```

In [18]:
```
s2[0:2] # by position
```

Out[18]:
```
a    2
b    4
dtype: int64
```

In [19]:
```
s2['a':'c'] # by label, the last index is INCLUDED!!!
```

Out[19]:
```
a    2
b    4
c    6
dtype: int64
```

In [20]:
```
s2.index
```

Out[20]:
```
Index(['a', 'b', 'c'], dtype='object')
```

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

In [21]:
```
s3= pd.Series(['a','b','c','d','e'])
s3
```

Out[21]:
```
0    a
1    b
2    c
3    d
4    e
dtype: object
```

In [22]:
```
s3.index
```

Out[22]:
```
RangeIndex(start=0, stop=5, step=1)
```

In [23]:
```
s3[0:2] #slicing -- this is confusing, although it is still by position
```

Out[23]:
```
0    a
1    b
dtype: object
```

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

In [24]:
```
s3.loc[0:2] # by label
```

Out[24]:
```
0    a
1    b
2    c
dtype: object
```

In [25]:
```
s3.iloc[0:2] # by position
```

Out[25]:
```
0    a
1    b
dtype: object
```

The same applies to DataFrame.

In [26]:
```
head
```

Out[26]:

	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	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	

5 rows × 21 columns

In [27]:
```
head.iloc[:3,:2]
```

Out[27]:

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

In [28]:
```
head.loc[:3,:'date' ]
```

Out[28]:

	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 [29]:
```
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 [ ]:
```
help(head.iloc)
```

In [31]:
```
head.loc[0,'price']
head.at[0,'price'] # .at can only access to one value
```

Out[31]:
```
221900.0
```

In [32]:
```
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 [33]:
```
head['date'] #same with head.date, indexing -column, no problem
```

Out[33]:
```
0    20141013T000000
1    20141209T000000
2    20150225T000000
3    20141209T000000
4    20150218T000000
Name: date, dtype: object
```

In [34]:
```
head[['date','price']] # fancy indexing -column, no problem
```

Out[34]:

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

In [35]:
```
head[['date']] # fancy indexing -column, no problem, get the dataframe instead of serie
```

Out[35]:

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

In [36]:
```
head[0:2] #slicing -- rows
```

Out[36]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfro
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfro
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	

2 rows × 21 columns

In [37]:
```
head['date':'price'] # this is wrong -- slicing cannot be applied to rows!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-4e474bdfffd7> 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__(self, ke
y)
   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_index_
sliceable(obj, key)
   2208     idx = obj.index
   2209     if isinstance(key, slice):
-> 2210         return idx._convert_slice_indexer(key, kind="getitem")
   2211
   2212     elif isinstance(key, str):

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py in _convert_slic
e_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_ind
exer(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 [38]:
```
head[:,'date':'price']# this is also wrong!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-963ada82415c> in <module>
----> 1 head[:,'date':'price']# this is also wrong!

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/frame.py in __getitem__(self, ke
y)
   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(self,
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 key
```

In [39]:

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-585d464c5f17> 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__(self, ke
y)
   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(self,
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 [40]:

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

Out[40]:

	date	price
1	20141209T000000	538000.0
2	20150225T000000	180000.0

In [45]:

```
head.loc[:,'date':'bedrooms'] # no problem for slicing in .loc
```

Out[45]:

	date	price	bedrooms
0	20141013T000000	221900.0	3

	date	price	bedrooms
1	20141209T000000	538000.0	3
2	20150225T000000	180000.0	2
3	20141209T000000	604000.0	4
4	20150218T000000	510000.0	3

In [48]:
```
head.loc[2,['date','bedrooms']] # fancy indexing is also supported in .loc
```

Out[48]:
```
date        20150225T000000
bedrooms                  2
Name: 2, dtype: object
```

In [52]:
```
states
```

Out[52]:

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

In [66]:
```
states.loc[:'New York', ['Area']]
```

Out[66]:

	Area
California	423967
Texas	695662
New York	141297

In [53]:
```
states['California':'Texas']
```

Out[53]:

	Population	Area
California	38332521	423967
Texas	26448193	695662

In [56]:
```
states['Population']
```

Out[56]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
```

```
Illinois      12882135
Name: Population, dtype: int64
```

In [58]:
```
states['California':'Texas','population'] # this is wrong, cannot do both!
```

Out[58]:
```
California    38332521
Texas         26448193
Name: Population, dtype: int64
```

In [60]:
```
states.loc['California':'Texas','Population']
```

Out[60]:
```
California    38332521
Texas         26448193
Name: Population, dtype: int64
```

In [61]:
```
states.loc['California':'Texas']
```

Out[61]:

	Population	Area
California	38332521	423967
Texas	26448193	695662

Boolean Selection

In [68]:
```
ind = states.Area>200000
ind
```

Out[68]:
```
California     True
Texas          True
New York      False
Florida       False
Illinois      False
Name: Area, dtype: bool
```

In [69]:
```
states[ind]
```

Out[69]:

	Population	Area
California	38332521	423967
Texas	26448193	695662

In [70]:
```
states[ind,'area'] # this is wrong!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-70-f5b87c24aa30> in <module>
----> 1 states[ind,'area'] # this is wrong!

~/opt/anaconda3/lib/python3.7/site-packages/pandas/core/frame.py in __getitem__(self, ke
y)
   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(self,
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: '(California     True
Texas          True
New York      False
Florida       False
Illinois      False
Name: Area, dtype: bool, 'area')' is an invalid key
```

In [72]:
```
states[ind]['Area']
```

Out[72]:
```
California    423967
Texas         695662
Name: Area, dtype: int64
```

In [74]:
```
states.loc[states.Area>200000,'Population'] # equivalently, states.loc[ind,'population'
```

Out[74]:
```
California    38332521
Texas         26448193
Name: Population, dtype: int64
```

In [75]:
```
states.iloc[ind.to_numpy(),1] # in iloc, the boolen should be the Numpy array
```

Out[75]:
```
California    423967
Texas         695662
Name: Area, dtype: int64
```

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 [76]:
```
house_price[house_price.loc[:,'bedrooms'].isin([2,4])]
```

Out[76]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
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 [77]:
```
house_price.loc[:,'bedrooms'].isin([2,4])
```

Out[77]:
```
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 [78]:
```
house_price[house_price['bedrooms'].isin([2,4])] # the same with column index
```

Out[78]:

	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	

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
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 [79]:
```
house_price[(house_price['bedrooms']==2)|(house_price['bedrooms']==4)] #equivalent way
```

Out[79]:

	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

Basic Manipulation

- Rename

In [80]:
```
states
```

Out[80]:

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

In [81]:
```

```

```
states_new = states.rename(columns = {"Population":"population","Area":"area"},index =
states_new
```

Out[81]:

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

In [82]:

```
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 [83]:
```
states
```

Out[83]:

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

In [84]:
```
states['density'] = states['Population']/states['Area'] # add new column
states
```

Out[84]:

	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

In [85]:
```
new_row = pd.DataFrame({'Population':7614893, 'Area':184827},index = ['Washington'])
new_row
```

Out[85]:

	Population	Area
Washington	7614893	184827

In [86]:
```
states_new = states.append(new_row)
states_new
```

Out[86]:

	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 [87]:
```
states_new.drop(index = "Washington",columns = "density",inplace = True)
states_new
```

Out[87]:

	Population	Area
California	38332521	423967
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 [88]:
```
states_new1 = pd.concat([states,new_row])
states_new1
```

Out[88]:

	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 [89]:
```
states_new
```

Out[89]:

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

In [90]:
```
pd.concat([states_new,states_new1.loc[:"Illinois","density"]],axis = 1)
```

Out[90]:

	Population	Area	density
California	38332521	423967	90.413926
Texas	26448193	695662	38.018740

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

In [91]:
```
help(pd.concat)
```

```
Help on function concat in module pandas.core.reshape.concat:

concat(objs: Union[Iterable[ForwardRef('NDFrame')], Mapping[Union[Hashable, NoneType], F
orwardRef('NDFrame')]], axis=0, join='outer', ignore_index: bool = False, keys=None, lev
els=None, names=None, verify_integrity: bool = False, sort: bool = False, copy: bool = T
rue) -> Union[ForwardRef('DataFrame'), ForwardRef('Series')]
    Concatenate pandas objects along a particular axis with optional set logic
    along the other axes.

    Can also add a layer of hierarchical indexing on the concatenation axis,
    which may be useful if the labels are the same (or overlapping) on
    the passed axis number.

    Parameters
    ----------
    objs : a sequence or mapping of Series or DataFrame objects
        If a mapping is passed, the sorted keys will be used as the `keys`
        argument, unless it is passed, in which case the values will be
        selected (see below). Any None objects will be dropped silently unless
        they are all None in which case a ValueError will be raised.
    axis : {0/'index', 1/'columns'}, default 0
        The axis to concatenate along.
    join : {'inner', 'outer'}, default 'outer'
        How to handle indexes on other axis (or axes).
    ignore_index : bool, default False
        If True, do not use the index values along the concatenation axis. The
        resulting axis will be labeled 0, ..., n - 1. This is useful if you are
        concatenating objects where the concatenation axis does not have
        meaningful indexing information. Note the index values on the other
        axes are still respected in the join.
    keys : sequence, default None
        If multiple levels passed, should contain tuples. Construct
        hierarchical index using the passed keys as the outermost level.
    levels : list of sequences, default None
        Specific levels (unique values) to use for constructing a
        MultiIndex. Otherwise they will be inferred from the keys.
    names : list, default None
        Names for the levels in the resulting hierarchical index.
    verify_integrity : bool, default False
        Check whether the new concatenated axis contains duplicates. This can
        be very expensive relative to the actual data concatenation.
    sort : bool, default False
        Sort non-concatenation axis if it is not already aligned when `join`
        is 'outer'.
        This has no effect when ``join='inner'``, which already preserves
        the order of the non-concatenation axis.

        .. versionchanged:: 1.0.0

           Changed to not sort by default.

    copy : bool, default True
        If False, do not copy data unnecessarily.
```

```
Returns
-------
object, type of objs
    When concatenating all ``Series`` along the index (axis=0), a
    ``Series`` is returned. When ``objs`` contains at least one
    ``DataFrame``, a ``DataFrame`` is returned. When concatenating along
    the columns (axis=1), a ``DataFrame`` is returned.

See Also
--------
Series.append : Concatenate Series.
DataFrame.append : Concatenate DataFrames.
DataFrame.join : Join DataFrames using indexes.
DataFrame.merge : Merge DataFrames by indexes or columns.

Notes
-----
The keys, levels, and names arguments are all optional.

A walkthrough of how this method fits in with other tools for combining
pandas objects can be found `here
<https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html>`__.

Examples
--------
Combine two ``Series``.

>>> s1 = pd.Series(['a', 'b'])
>>> s2 = pd.Series(['c', 'd'])
>>> pd.concat([s1, s2])
0    a
1    b
0    c
1    d
dtype: object

Clear the existing index and reset it in the result
by setting the ``ignore_index`` option to ``True``.

>>> pd.concat([s1, s2], ignore_index=True)
0    a
1    b
2    c
3    d
dtype: object

Add a hierarchical index at the outermost level of
the data with the ``keys`` option.

>>> pd.concat([s1, s2], keys=['s1', 's2'])
s1  0    a
    1    b
s2  0    c
    1    d
dtype: object

Label the index keys you create with the ``names`` option.

>>> pd.concat([s1, s2], keys=['s1', 's2'],
...           names=['Series name', 'Row ID'])
Series name  Row ID
s1           0         a
             1         b
s2           0         c
```

```
            1        d
dtype: object

Combine ``DataFrame`` objects with identical columns.

>>> df1 = pd.DataFrame([['a', 1], ['b', 2]],
...                    columns=['letter', 'number'])
>>> df1
  letter  number
0      a       1
1      b       2
>>> df2 = pd.DataFrame([['c', 3], ['d', 4]],
...                    columns=['letter', 'number'])
>>> df2
  letter  number
0      c       3
1      d       4
>>> pd.concat([df1, df2])
  letter  number
0      a       1
1      b       2
0      c       3
1      d       4

Combine ``DataFrame`` objects with overlapping columns
and return everything. Columns outside the intersection will
be filled with ``NaN`` values.

>>> df3 = pd.DataFrame([['c', 3, 'cat'], ['d', 4, 'dog']],
...                    columns=['letter', 'number', 'animal'])
>>> df3
  letter  number animal
0      c       3    cat
1      d       4    dog
>>> pd.concat([df1, df3], sort=False)
  letter  number animal
0      a       1    NaN
1      b       2    NaN
0      c       3    cat
1      d       4    dog

Combine ``DataFrame`` objects with overlapping columns
and return only those that are shared by passing ``inner`` to
the ``join`` keyword argument.

>>> pd.concat([df1, df3], join="inner")
  letter  number
0      a       1
1      b       2
0      c       3
1      d       4

Combine ``DataFrame`` objects horizontally along the x axis by
passing in ``axis=1``.

>>> df4 = pd.DataFrame([['bird', 'polly'], ['monkey', 'george']],
...                    columns=['animal', 'name'])
>>> pd.concat([df1, df4], axis=1)
  letter  number  animal    name
0      a       1    bird   polly
1      b       2  monkey  george

Prevent the result from including duplicate index values with the
``verify_integrity`` option.
```

```
>>> df5 = pd.DataFrame([1], index=['a'])
>>> df5
   0
a  1
>>> df6 = pd.DataFrame([2], index=['a'])
>>> df6
   0
a  2
>>> pd.concat([df5, df6], verify_integrity=True)
Traceback (most recent call last):
    ...
ValueError: Indexes have overlapping values: ['a']
```

- Merge: "Concat by Value"

In [92]:
```
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 [93]:
```
df1
```

Out[93]:

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

In [94]:
```
df2
```

Out[94]:

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

In [95]:
```
pd.concat([df1,df2])
```

Out[95]:

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

	employee	group	hire_date
0	Lisa	NaN	2004.0
1	Bob	NaN	2008.0
2	Jake	NaN	2012.0
3	Sue	NaN	2014.0

In [96]:
```
pd.concat([df1,df2],axis=1)
```

Out[96]:

	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 [97]:
```
pd.merge(df1,df2)
```

Out[97]:

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

In [98]:
```
df3 = pd.merge(df1,df2,on="employee")
df3
```

Out[98]:

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

In [99]:
```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
df4
```

Out[99]:

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido

	group	supervisor
2	HR	Steve

In [100…

```
pd.merge(df3,df4)
```

Out[100…

	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