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: numpy>=1.17.3 in e:\programdata\anaconda3\lib\site-packag
es (from pandas) (1.20.1)
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: 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 [3]:
```
pd.__version__
```

Out[3]:
```
'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 [9]:
```
s1 = pd.Series([2, 4, 6])
print(s1)
```

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

In [6]:
```
type(s1)
```

Out[6]:
```
pandas.core.series.Series
```

In [7]:
```
s1.index
```

Out[7]:
```
RangeIndex(start=0, stop=3, step=1)
```

In [10]:
```
s2 = pd.Series([2, 4, 6],index = ['a','b','c'])
```

In [11]:
```
s2
```

Out[11]:
```
a    2
b    4
c    6
dtype: int64
```

In [15]:
```
s2_num = s2.values # change to Numpy -- can be view instead of copy if the elements are
s2_num
```

Out[15]:
```
array([2, 4, 6], dtype=int64)
```

In [16]:
```
np.shares_memory(s2_num,s2)
```

Out[16]:
```
True
```

In [17]:
```
s2_num_copy = s2.to_numpy(copy = True) # more recommended in new version of Pandas -- c
np.shares_memory(s2_num_copy,s2)
```

Out[17]:
```
False
```

Selection by position -- similar to Numpy array!

In [25]:
```
s2[0:2]
```

Out[25]:
```
a    2
b    4
dtype: int64
```

Selection by index (label)

In [26]:
```
s2['a']
```

Out[26]:
```
2
```

In [27]:
```
s2[['a','c']]
```

Out[27]:
```
a    2
c    6
dtype: int64
```

`Series` and Python Dictionary

In [28]:
```
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[28]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

In [34]:
```
population_dict['Texas'] # key and value
```

Out[34]:
```
26448193
```

In [35]:
```
population['Texas']
```

Out[35]:
```
26448193
```

In [36]:
```
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[36]:
```
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 [37]:
```
states = pd.DataFrame({'Population': population,
                       'Area': area}) # variable names
states
```

Out[37]:

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

In [38]:
```
type(states)
```

Out[38]:
```
pandas.core.frame.DataFrame
```

In [39]:
```
states.index
```

Out[39]:
```
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 [43]:
```
type(states['Area'])
```

Out[43]:
```
pandas.core.series.Series
```

In [44]:
```
random = pd.DataFrame(np.random.rand(3, 2),columns=['foo', 'bar'],index=['a', 'b', 'c']
random
```

Out[44]:

	foo	bar
a	0.430201	0.372644
b	0.673136	0.048943
c	0.066612	0.621519

In [45]:
```
random.T
```

Out[45]:

	a	b	c
foo	0.430201	0.673136	0.066612
bar	0.372644	0.048943	0.621519

Creating DataFrame from Files

In [46]:
```
house_price = pd.read_csv('kc_house_data.csv')
house_price
```

Out[46]:

	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 [47]:
```
house_price.shape # dimension of the data
```

Out[47]:
```
(21613, 21)
```

In [48]:
```
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 [52]:
```
house_price.head(3) # show the head lines
```

Out[52]:

	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 [54]:
```
house_price.sample(5) # show the random samples
```

Out[54]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wat
21539	8902000201	20150219T000000	338500.0	3	2.25	1333	1470	3.0	
15879	8807300130	20141217T000000	330000.0	3	1.00	910	10240	1.0	
15684	8731000010	20140515T000000	343000.0	4	1.75	2290	10290	1.0	
17048	7889600080	20150219T000000	208000.0	3	1.00	1050	6240	1.0	
6414	9407600250	20150327T000000	211000.0	3	2.00	1060	7412	1.0	

5 rows × 21 columns

In [55]:
```
house_price.describe() # descriptive statistics
```

Out[55]:

	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 [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 [56]:
```
head = house_price.head()
head.to_csv('head.csv')
```

In [57]:
```
head
```

Out[57]:

	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 [58]:
```
head.sort_values(by='price')
```

Out[58]:

	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 [59]:
```
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 [60]:
```
head.to_numpy()
```

Out[60]:
```
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 [61]:
```
s2
```

Out[61]:
```
a    2
b    4
c    6
dtype: int64
```

In [62]:
```
s2[0:2] # by position, last index not included
```

Out[62]:
```
a    2
b    4
dtype: int64
```

In [63]:
```
s2['a':'c'] # by label, the last index is INCLUDED!!!
```

Out[63]:
```
a    2
b    4
c    6
dtype: int64
```

In [64]:
```
s2.index
```

Out[64]:
```
Index(['a', 'b', 'c'], dtype='object')
```

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

In [65]:
```
s3= pd.Series(['a','b','c','d','e'])
s3
```

Out[65]:
```
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 [69]:
```
s3.iloc[0:2] # by position.
```

Out[69]:
```
0    a
1    b
dtype: object
```

The same applies to DataFrame.

In [72]:
```
head
```

Out[72]:

	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	

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfro
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 [73]:
```
head.iloc[:3,:2]
```

Out[73]:

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

In [74]:
```
head.loc[:3,:'date' ]
```

Out[74]:

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

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

In [80]:
```
head.loc[0,'price']
```

Out[80]: 221900.0

In [79]:
```
head.at[0,'price'] # .at can only access to one value
```

Out[79]: 221900.0

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

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 [81]:
```
head['date'] #same with head.date, indexing -column, no problem
```

Out[81]:
```
0    20141013T000000
1    20141209T000000
2    20150225T000000
3    20141209T000000
4    20150218T000000
Name: date, dtype: object
```

In [82]:
```
head[['date','price']] # fancy indexing -column, no problem
```

Out[82]:

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

In [83]:
```
head[['date']] # fancy indexing -column, no problem, get the dataframe instead of serie
```

Out[83]:

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

In [84]:
```
head[0:2] #slicing -- rows
```

Out[84]:

	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 [85]:
```
head['date':'price'] # this is wrong -- slicing cannot be applied to rows!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-85-4e474bdfffd7> in <module>
----> 1 head['date':'price'] # this is wrong -- slicing cannot be applied to rows!

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 [86]:
```
head[:,'date':'price']# this is also wrong!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-963ada82415c> 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 [87]:

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-87-585d464c5f17> 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 [90]:

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

Out[90]:

	date	price
1	20141209T000000	538000.0
2	20150225T000000	180000.0

In [91]:

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

Out[91]:

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

In [92]:
```
head.loc[2,['date','bedrooms']] # fancy indexing is also supported in .loc
```

Out[92]:
```
date        20150225T000000
bedrooms                  2
Name: 2, dtype: object
```

In [93]:
```
states
```

Out[93]:

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

In [94]:
```
states.loc[:'New York', ['Area']]
```

Out[94]:

	Area
California	423967
Texas	695662
New York	141297

In [95]:
```
states['California':'Texas']
```

Out[95]:

	Population	Area
California	38332521	423967
Texas	26448193	695662

In [96]:
```
states['Population']
```

Out[96]:
```
California    38332521
Texas         26448193
New York      19651127
```

```
Florida       19552860
Illinois      12882135
Name: Population, dtype: int64
```

In [97]:
```
states['California':'Texas','population'] # this is wrong, cannot do both!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-97-048ac2c79b68> in <module>
----> 1 states['California':'Texas','population'] # this is 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('California', 'Texas', None), 'population')' is an invalid key
```

In [98]:
```
states.loc['California':'Texas','Population']
```

Out[98]:
```
California    38332521
Texas         26448193
Name: Population, dtype: int64
```

In [99]:
```
states.loc['California':'Texas']
```

Out[99]:

	Population	Area
California	38332521	423967
Texas	26448193	695662

Boolean Selection

In [100…
```
ind = states.Area>200000
ind
```

Out[100…
```
California     True
Texas          True
New York      False
Florida       False
```

```
Illinois      False
Name: Area, dtype: bool
```

In [101…
```
states[ind]
```

Out[101…

	Population	Area
California	38332521	423967
Texas	26448193	695662

In [102…
```
states[ind,'area'] # this is wrong!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-102-f5b87c24aa30> in <module>
----> 1 states[ind,'area'] # this is 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: '(California     True
Texas          True
New York      False
Florida       False
Illinois      False
Name: Area, dtype: bool, 'area')' is an invalid key
```

In [103…
```
states[ind]['Area']
```

Out[103…
```
California    423967
Texas         695662
Name: Area, dtype: int64
```

In [105…
```
states.loc[states.Area>200000,'Population'] # equivalently, states.loc[ind,'population'
```

Out[105…
```
California    38332521
Texas         26448193
Name: Population, dtype: int64
```

In [106…
```
states.iloc[ind.to_numpy(),1] # in iloc, the boolean should be the Numpy array
```

Out[106…
```
California    423967
Texas         695662
Name: Area, dtype: int64
```

In [107…
```
random
```

Out[107…

	foo	bar
a	0.430201	0.372644
b	0.673136	0.048943
c	0.066612	0.621519

In [108…
```
random[random['foo']>0.6]
```

Out[108…

	foo	bar
b	0.673136	0.048943

In [109…
```
house_price
```

Out[109…

	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

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

In [110…
```
house_price[house_price.loc[:,'bedrooms'].isin([2,4])]   #either 2 bedrooms or 4 bedroo
```

Out[110…

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa

	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 [111…

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

Out[111…

```
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 [112…

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

Out[112…

	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	

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
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 [113…

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

Out[113…

	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 [114…

```
states
```

Out[114…

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

In [115…
```
states_new = states.rename(columns = {"Population":"population","Area":"area"},index =
states_new
```

Out[115…

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

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

- Append/Drop

In [116…
```
states
```

Out[116…

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

In [117…
```
states['density'] = states['Population']/states['Area'] # add new column
states
```

Out[117…

	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 [118…
```
new_row = pd.DataFrame({'Population':7614893, 'Area':184827},index = ['Washington'])
new_row
```

Out[118…

	Population	Area
Washington	7614893	184827

In [119…
```
states_new = states.append(new_row)
states_new
```

Out[119…

	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 [120…
```
states_new.drop(index = "Washington",columns = "density",inplace = True)
states_new
```

Out[120…

	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 [122…
```
states_new1 = pd.concat([states,new_row])
states_new1
```

Out[122…

	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 [123…
```
states_new #the pd.concat() function does not affect the original states_new
```

Out[123...

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

In [128...

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

Out[128...

	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 [ ]:

```
help(pd.concat)
```

- Merge: "Concat by Value"

In [129...

```
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 [132...

```
df1
```

Out[132...

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

In [133...

```
df2
```

Out[133...

	employee	hire_date
0	Lisa	2004

	employee	hire_date
1	Bob	2008
2	Jake	2012
3	Sue	2014

In [134…
```
pd.concat([df1,df2])
```

Out[134…

	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
3	Sue	NaN	2014.0

In [135…
```
pd.concat([df1,df2],axis=1)
```

Out[135…

	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 [136…
```
pd.merge(df1,df2)
```

Out[136…

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

In [137…
```
df3 = pd.merge(df1,df2,on="employee")
df3
```

Out[137…

	employee	group	hire_date

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

In [139…

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

Out[139…

	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