Pandas

In [143]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Type *Markdown* and LaTeX: α^2

- pandas contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python.
- pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib.
- pandas adopts significant parts of NumPy's idiomatic style of array-based computing, especially array-based functions and a preference for data processing without *for* loops.
- While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with *tabular* or *heterogeneous* data. NumPy, by contrast, is best suited for working with *homogeneous numerical* array data.

Introduction to pandas Data Structures

Series

A **Series** is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its *index*. The simplest Series is formed from only an array of data:

In [144]:
```
obj = pd.Series([4, 7, -5, 3])
```

In [145]:
```
obj
```

Out[145]:
```
0    4
1    7
2   -5
3    3
dtype: int64
```

Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created. You can get the array representation and index object of the Series via its **values** and **index** attributes, respectively:

Processing math: 100%

In [146]:
```
obj.values
```

Out[146]:
```
array([ 4,  7, -5,  3], dtype=int64)
```

In [147]:
```
obj.index
```

Out[147]:
```
RangeIndex(start=0, stop=4, step=1)
```

Often it will be desirable to create a Series with an **index** identifying each data point with a label:

In [148]:
```
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
```

In [149]:
```
obj2
```

Out[149]:
```
d    4
b    7
a   -5
c    3
dtype: int64
```

In [150]:
```
obj2.index
```

Out[150]:
```
Index(['d', 'b', 'a', 'c'], dtype='object')
```

You can use labels in the index when selecting single values or a set of values:

In [151]:
```
obj2['a']
```

Out[151]:
```
-5
```

In [152]:
```
obj2['d'] = 6
```

In [153]:
```
obj2
```

Out[153]:
```
d    6
b    7
a   -5
c    3
dtype: int64
```

In [154]:
```
obj2[['c','a','d']]
```

Out[154]:
```
c    3
a   -5
d    6
dtype: int64
```

Processing math: 100%

Using NumPy functions or NumPy-like operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

In [155]:
```
obj2[obj2 > 0]
```

Out[155]:
```
d    6
b    7
c    3
dtype: int64
```

In [156]:
```
obj2 * 2
```

Out[156]:
```
d    12
b    14
a   -10
c     6
dtype: int64
```

In [157]:
```
np.exp(obj2)
```

Out[157]:
```
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Another way to think about a Series is as a fixed-length, *ordered dict*, as it is a mapping of index values to data values. It can be used in many contexts where you might use a **dict**:

In [158]:
```
'b' in obj2
```

Out[158]:
```
True
```

In [159]:
```
'e' in obj2
```

Out[159]:
```
False
```

You can create a Series from it by passing the dict:

In [160]:
```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

In [161]:
```
obj3 = pd.Series(sdata)
```

Processing math: 100%

In [162]:
```
obj3
```

Out[162]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

When you are only passing a *dict*, the index in the resulting Series will have the *dict's keys in sorted order*. You can override this by passing the dict keys in the order you want them to appear in the resulting Series:

In [163]:
```
states = ['California', 'Ohio', 'Oregon', 'Texas']
```

In [164]:
```
obj4 = pd.Series(sdata, index=states)
```

In [165]:
```
obj4
```

Out[165]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

The **isnull** and **notnull** functions in pandas should be used to detect missing data:

In [166]:
```
pd.isnull(obj4)
```

Out[166]:
```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

In [167]:
```
pd.notnull(obj4)
```

Out[167]:
```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series also has these as instance methods:

Processing math: 100%

In [168]:
```
obj4.isnull()
```

Out[168]:
```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

In [169]:
```
obj4.notnull()
```

Out[169]:
```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

A useful Series feature for many applications is that it *automatically aligns by index label* in arithmetic operations:

In [170]:
```
obj3
```

Out[170]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

In [171]:
```
obj4
```

Out[171]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

In [172]:
```
obj3 + obj4
```

Out[172]:
```
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

Both the *Series object itself* and its *index* have a **name** attribute, which integrates with other key areas of pandas functionality:

Processing math: 100%

In [173]:
```
obj4
```

Out[173]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

In [174]:
```
obj4.name = 'population'
```

In [175]:
```
obj4.index.name = 'state'
```

In [176]:
```
obj4
```

Out[176]:
```
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

A Series's index can be altered in-place by assignment:

In [177]:
```
obj
```

Out[177]:
```
0    4
1    7
2   -5
3    3
dtype: int64
```

In [178]:
```
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

In [179]:
```
obj
```

Out[179]:
```
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

DataFrame

- A DataFrame represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.).
- The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index.

Processing math: 100%

- The data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays.
- While a DataFrame is physically two-dimensional, you can use it to represent higher dimensional data in a tabular format using hierarchical indexing.

There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays:

In [180]:
```
data = {'state':
 ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
  'year': [2000, 2001, 2002, 2001, 2002, 2003],
  'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

In [181]:
```
frame
```

Out[181]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

For large DataFrames, the **head** method selects only the first five rows:

In [182]:
```
frame.head()
```

Out[182]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

In [183]:
```
frame.head(2)
```

Out[183]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7

Processing math: 100%

If you specify a sequence of **columns**, the DataFrame's columns will be arranged in that order:

In [184]:
```
pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

Out[184]:

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

If you pass a column that isn't contained in the dict, it will appear with missing values in the result:

In [185]:
```
data
```

Out[185]:
```
{'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
 'year': [2000, 2001, 2002, 2001, 2002, 2003],
 'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
```

In [186]:
```
frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
index=['one', 'two', 'three', 'four','five', 'six'])
```

In [187]:
```
frame2
```

Out[187]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN
six	2003	Nevada	3.2	NaN

A column in a DataFrame can be retrieved *as a Series* either by *dict-like* notation or by *attribute*:

Processing math: 100%

In [188]:
```
frame2['state']
```

Out[188]:
```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

In [189]:
```
frame2.state
```

Out[189]:
```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

Note that the returned Series have the *same index* as the DataFrame, and their name attribute has been appropriately set.

The column returned from indexing a DataFrame is a view on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied with the Series's *copy* method.

Rows can also be retrieved by position or name with the special **.loc** attribute:

In [190]:
```
frame2.loc['three']
```

Out[190]:
```
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

Columns can be modified by assignment. For example, the empty 'debt' column could be assigned a scalar value or an array of values:

In [191]:
```
frame2['debt'] = 16.5
```

Processing math: 100%

In [192]:
```
frame2
```

Out[192]:

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5
six	2003	Nevada	3.2	16.5

In [193]:
```
frame2['debt'] = np.arange(6.)
```

In [194]:
```
frame2
```

Out[194]:

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0
six	2003	Nevada	3.2	5.0

When you are assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any holes:

In [195]:
```
val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
```

In [196]:
```
frame2['debt'] = val
```

Processing math: 100%

In [197]:
```
frame2
```

Out[197]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

Assigning a column that doesn't exist will create a new column. The **del** keyword will delete columns as with a dict.

In [198]:
```
frame2['eastern'] = frame2.state == 'Ohio'
```

In [199]:
```
frame2
```

Out[199]:

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False
six	2003	Nevada	3.2	NaN	False

In [200]:
```
del frame2['eastern']
```

In [201]:
```
frame2.columns
```

Out[201]:
```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Processing math: 100%

In [202]:
```
frame2
```

Out[202]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

New columns cannot be created with the frame2.eastern syntax.

Another common form of data is a nested dict of dicts:

In [203]:
```
pop = {'Nevada': {2001: 2.4, 2002: 2.9},
        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

pandas will interpret the *outer dict* keys as the *columns* and the *inner keys* as the row indices:

In [204]:
```
frame3 = pd.DataFrame(pop)
```

In [205]:
```
frame3
```

Out[205]:

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

You can transpose the DataFrame (swap rows and columns) with similar syntax to a NumPy array:

In [206]:
```
frame3.T
```

Out[206]:

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

The keys in the inner dicts are combined and sorted to form the index in the result. This isn't true if an explicit index is specified:

Processing math: 100%

In [207]:
```
##pop = {'Nevada': {2001: 2.4, 2002: 2.9},         'Ohio': {2000: 1.5, 2001: 1.7,
```

In [208]:
```
## pd.DataFrame(pop, index=['a', 'b'])
```

Dicts of Series are treated in much the same way:

In [209]:
```
pdata = {'Ohio': frame3['Ohio'][:-1], 'Nevada': frame3['Nevada'][:2]}
```

In [210]:
```
pd.DataFrame(pdata)
```

Out[210]:

	Ohio	Nevada
2000	1.5	NaN
2001	1.7	2.4

Table 5-1. Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column; indexes from each Series are unioned together to form the result's row index if no explicit index is passed
dict of dicts	Each inner dict becomes a column; keys are unioned to form the row index as in the "dict of Series" case
List of dicts or Series	Each item becomes a row in the DataFrame; union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

If a DataFrame's index and columns have their *name* attributes set, these will also be displayed:

In [211]:
```
frame3.index.name = 'year'; frame3.columns.name = 'state'
```

Processing math: 100%

In [212]:
```
frame3
```

Out[212]:

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

As with Series, the **values** attribute returns the data contained in the DataFrame as a two-dimensional ndarray:

In [213]:
```
frame3.values
```

Out[213]:
```
array([[nan, 1.5],
       [2.4, 1.7],
       [2.9, 3.6]])
```

If the DataFrame's columns are different dtypes, the dtype of the values array will be chosen to accommodate all of the columns:

In [214]:
```
frame2.values
```

Out[214]:
```
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

Index Objects

pandas's *Index* objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index:

In [215]:
```
obj = pd.Series(range(3), index=['a', 'b', 'c'])
```

In [216]:
```
obj
```

Out[216]:
```
a    0
b    1
c    2
dtype: int64
```

Processing math: 100%

In [217]:
```
index = obj.index
```

In [218]:
```
index
```

Out[218]:
```
Index(['a', 'b', 'c'], dtype='object')
```

In [219]:
```
index[1:]
```

Out[219]:
```
Index(['b', 'c'], dtype='object')
```

Index objects are immutable and thus can't be modified by the user:

Immutability makes it safer to share Index objects among data structures:

In [220]:
```
labels = pd.Index(np.arange(3))
```

In [221]:
```
labels
```

Out[221]:
```
Int64Index([0, 1, 2], dtype='int64')
```

In [222]:
```
obj2 = pd.Series([1.5, -2.5, 0], index=labels)
```

In [223]:
```
obj2.index is labels
```

Out[223]:
```
True
```

In addition to being array-like, an Index also behaves like a fixed-size set:

In [224]:
```
frame3
```

Out[224]:

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

In [225]:
```
frame3.columns
```

Out[225]:
```
Index(['Nevada', 'Ohio'], dtype='object', name='state')
```

In [226]:
```
'Ohio' in frame3.columns
```

Out[226]:
```
True
```

Processing math: 100%

Unlike Python sets, a pandas Index can contain duplicate labels:

In [227]:
```
dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
```

In [228]:
```
dup_labels
```

Out[228]:
```
Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Selections with duplicate labels will select all occurrences of that label.

Each Index has a number of methods and properties for set logic, which answer other common questions about the data it contains.

Table 5-2. Some Index methods and properties

Method	Description
append	Concatenate with additional Index objects, producing a new Index
difference	Compute set difference as an Index
intersection	Compute set intersection
union	Compute set union
isin	Compute boolean array indicating whether each value is contained in the passed collection
delete	Compute new Index with element at index i deleted
drop	Compute new Index by deleting passed values
insert	Compute new Index by inserting element at index i
is_monotonic	Returns True if each element is greater than or equal to the previous element
is_unique	Returns True if the Index has no duplicate values
unique	Compute the array of unique values in the Index

Essential Functionality

Reindexing

An important method on pandas objects is **reindex**, which means to create a new object with the data conformed to a new index. Consider an example:

In [229]:
```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
```

Processing math: 100%

In [230]:
```
obj
```

Out[230]:
```
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

Calling **reindex** on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

In [231]:
```
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
```

In [232]:
```
obj2
```

Out[232]:
```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

For ordered data like time series, it may be desirable to do some interpolation or fill- ing of values when reindexing. The method option allows us to do this, using a method such as **ffill**, which forward-fills the values:

In [233]:
```
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
```

In [234]:
```
obj3
```

Out[234]:
```
0      blue
2    purple
4    yellow
dtype: object
```

In [235]:
```
obj3.reindex(range(6), method='ffill')
```

Out[235]:
```
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

With DataFrame, reindex can alter either the (row) index, columns, or both. When passed only a sequence, it reindexes the rows in the result:

Processing math: 100%

In [236]:
```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
index=['a', 'c', 'd'],
columns=['Ohio', 'Texas', 'California'])
```

In [237]:
```
frame
```

Out[237]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

In [238]:
```
frame2 = frame.reindex(['a', 'b', 'c', 'd'])
```

In [239]:
```
frame2
```

Out[239]:

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

The columns can be reindexed with the **columns** keyword:

In [240]:
```
states = ['Texas', 'Utah', 'California']
```

In [241]:
```
frame.reindex(columns=states)
```

Out[241]:

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

You can reindex more succinctly by label-indexing with loc, and many users prefer to use it exclusively:

Processing math: 100%

In [242]:
```
frame.loc[['a', 'b', 'c', 'd'], states]
```

```
C:\Users\masoud\Anaconda3\envs\Iris_2\lib\site-packages\ipykernel_launcher.py:
1: FutureWarning:
Passing list-likes to .loc or [] with any missing label will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
https://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate-loc-reinde
x-listlike (https://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecat
e-loc-reindex-listlike)
  """Entry point for launching an IPython kernel.
```

Out[242]:

	Texas	Utah	California
a	1.0	NaN	2.0
b	NaN	NaN	NaN
c	4.0	NaN	5.0
d	7.0	NaN	8.0

Table 5-3. reindex function arguments

Argument	Description
`index`	New sequence to use as index. Can be Index instance or any other sequence-like Python data structure. An Index will be used exactly as is without any copying.
`method`	Interpolation (fill) method; `'ffill'` fills forward, while `'bfill'` fills backward.
`fill_value`	Substitute value to use when introducing missing data by reindexing.
`limit`	When forward- or backfilling, maximum size gap (in number of elements) to fill.
`tolerance`	When forward- or backfilling, maximum size gap (in absolute numeric distance) to fill for inexact matches.
`level`	Match simple Index on level of MultiIndex; otherwise select subset of.
`copy`	If `True`, always copy underlying data even if new index is equivalent to old index; if `False`, do not copy the data when the indexes are equivalent.

Dropping Entries from an Axis

drop method will return a new object with the indicated value or values deleted from an axis:

In [243]:
```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

In [244]:
```
obj
```

Out[244]:
```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

Processing math: 100%

In [245]:
```
new_obj = obj.drop('c')
```

In [246]:
```
new_obj
```

Out[246]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

In [247]:
```
obj.drop(['d', 'c'])
```

Out[247]:
```
a    0.0
b    1.0
e    4.0
dtype: float64
```

With DataFrame, index values can be deleted from either axis. To illustrate this, we first create an example DataFrame:

In [248]:
```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
index=['Ohio', 'Colorado', 'Utah', 'New York'],
columns=['one', 'two', 'three', 'four'])
```

In [249]:
```
data
```

Out[249]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [250]:
```
data.drop(['Colorado', 'Ohio'])
```

Out[250]:

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

You can drop values from the columns by passing *axis=1* or *axis='columns'*:

Processing math: 100%

In [251]:
```
data.drop('two', axis=1)
```

Out[251]:

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

In [252]:
```
data.drop(['two', 'four'], axis='columns')
```

Out[252]:

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

Many functions, like drop, which modify the size or shape of a Series or DataFrame, can manipulate an object **in-place** without returning a new object:

In [253]:
```
obj.drop('c', inplace=True)
```

In [254]:
```
obj
```

Out[254]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

Indexing, Selection, and Filtering

Series indexing (obj[...]) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers. Here are some examples of this:

In [255]:
```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
```

In [256]:
```
obj['b']
```

Out[256]:
```
1.0
```

Processing math: 100%

In [257]:
```
obj[1]
```

Out[257]:
```
1.0
```

In [258]:
```
obj[2:4]
```

Out[258]:
```
c    2.0
d    3.0
dtype: float64
```

In [259]:
```
obj[['b', 'a', 'd']]
```

Out[259]:
```
b    1.0
a    0.0
d    3.0
dtype: float64
```

In [260]:
```
obj[[1, 3]]
```

Out[260]:
```
b    1.0
d    3.0
dtype: float64
```

In [261]:
```
obj[obj < 2]
```

Out[261]:
```
a    0.0
b    1.0
dtype: float64
```

Slicing with labels behaves differently than normal Python slicing in that the endpoint is **inclusive**:

In [262]:
```
obj['b':'c']
```

Out[262]:
```
b    1.0
c    2.0
dtype: float64
```

In [263]:
```
obj['b':'c'] = 5
```

In [264]:
```
obj
```

Out[264]:
```
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

Processing math: 100%

In [265]:
```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
index=['Ohio', 'Colorado', 'Utah', 'New York'],
columns=['one', 'two', 'three', 'four'])
```

In [266]:
```
data
```

Out[266]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [267]:
```
data['two']
```

Out[267]:
```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int32
```

In [268]:
```
data[['three', 'one']]
```

Out[268]:

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

Indexing like this has a few special cases. First, slicing or selecting data with a boolean array:

In [269]:
```
data[:2]
```

Out[269]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

Processing math: 100%

In [270]:
```
data[data['three'] > 5]
```

Out[270]:

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

The row selection syntax data[:2] is provided as a convenience. Passing a single element or a list to the [] operator selects *columns*.

Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

In [271]:
```
data < 5
```

Out[271]:

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

In [272]:
```
data[data < 5] = 0
```

In [273]:
```
data
```

Out[273]:

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Selection with loc and iloc

loc and **iloc** enable you to select a subset of the rows and columns from a DataFrame with NumPy-like notation using either axis labels (loc) or integers (iloc).

In [274]:
```
data.loc['Colorado', ['two', 'three']]
```

Out[274]:
```
two      5
three    6
Name: Colorado, dtype: int32
```

In [275]:
```
data.iloc[2, [3, 0, 1]]
```

Out[275]:
```
four    11
one      8
two      9
Name: Utah, dtype: int32
```

In [276]:
```
data.iloc[2]
```

Out[276]:
```
one       8
two       9
three    10
four     11
Name: Utah, dtype: int32
```

In [277]:
```
data.iloc[[1, 2], [3, 0, 1]]
```

Out[277]:

	four	one	two
Colorado	7	0	5
Utah	11	8	9

Both indexing functions work with slices in addition to single labels or lists of labels:

In [278]:
```
data.loc[:'Utah', 'two']
```

Out[278]:
```
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int32
```

In [279]:
```
data.iloc[:, :3]
```

Out[279]:

	one	two	three
Ohio	0	0	0
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

In [280]:
```
data.iloc[:, :3][data.three > 5]
```

Out[280]:

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

Indexing options with DataFrame

Type	Notes
`df[val]`	Select single column or sequence of columns from the DataFrame; special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion)
`df.loc[val]`	Selects single row or subset of rows from the DataFrame by label
`df.loc[:, val]`	Selects single column or subset of columns by label
`df.loc[val1, val2]`	Select both rows and columns by label
`df.iloc[where]`	Selects single row or subset of rows from the DataFrame by integer position
`df.iloc[:, where]`	Selects single column or subset of columns by integer position
`df.iloc[where_i, where_j]`	Select both rows and columns by integer position
`df.at[label_i, label_j]`	Select a single scalar value by row and column label
`df.iat[i, j]`	Select a single scalar value by row and column position (integers)
`reindex` method	Select either rows or columns by labels
`get_value, set_value` methods	Select single value by row and column label

Integer Indexes

Working with pandas objects indexed by integers is something that often trips up new users due to some differences with indexing semantics on built-in Python data structures like lists and tuples. For example, you might not expect the following code to generate an error:

In [281]:
```
ser = pd.Series(np.arange(3.))
```

In [282]:
```
ser
```

Out[282]:
```
0    0.0
1    1.0
2    2.0
dtype: float64
```

In [283]:
```
#ser[-1] error
```

Processing math: 100%

In [284]:
```
ser.iloc[-1]
```

Out[284]:
```
2.0
```

On the other hand, with a non-integer index, there is no potential for ambiguity:

In [285]:
```
ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])
```

In [286]:
```
ser2[-1]
```

Out[286]:
```
2.0
```

To keep things consistent, if you have an axis index containing integers, data selection will always be label-oriented. For more precise handling, use loc (for labels) or iloc (for integers):

In [287]:
```
ser[:1]
```

Out[287]:
```
0    0.0
dtype: float64
```

In [288]:
```
ser.loc[:1]
```

Out[288]:
```
0    0.0
1    1.0
dtype: float64
```

In [289]:
```
ser.iloc[:1]
```

Out[289]:
```
0    0.0
dtype: float64
```

Arithmetic and Data Alignment

An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes. When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. For users with database experience, this is similar to an automatic outer join on the index labels. Let’s look at an example:

In [290]:
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
```

Processing math: 100%

In [291]:
```
s1
```

Out[291]:
```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

In [293]:
```
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
```

In [294]:
```
s2
```

Out[294]:
```
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

In [295]:
```
s1 + s2
```

Out[295]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

In the case of DataFrame, alignment is performed on both the rows and the columns:

In [296]:
```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'), index=['Ol
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'), index=['U
```

In [297]:
```
df1
```

Out[297]:

	b	c	d
Ohio	0.0	1.0	2.0
Texas	3.0	4.0	5.0
Colorado	6.0	7.0	8.0

Processing math: 100%

In [298]:
```
df2
```

Out[298]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [299]:
```
df1 + df2
```

Out[299]:

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

Arithmetic methods with fill values

In arithmetic operations between differently indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other:

In [300]:
```
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))
```

In [301]:
```
df1
```

Out[301]:

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

Processing math: 100%

In [302]:
```
df2
```

Out[302]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

In [303]:
```
df2.loc[1, 'b'] = np.nan
```

In [304]:
```
df2
```

Out[304]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	NaN	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

In [305]:
```
df1 + df2
```

Out[305]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	NaN	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

Using the **add** method on df1, I pass df2 and an argument to **fill_value**:

In [306]:
```
df1.add(df2, fill_value=0)
```

Out[306]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	5.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

See Table 5-5 for a listing of Series and DataFrame methods for arithmetic. Each of them has a counterpart, starting with the letter *r*, that has arguments flipped. So these two statements are equivalent:

Processing math: 100%

In [308]:
```
1 / df1
```

Out[308]:

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250000	0.200000	0.166667	0.142857
2	0.125000	0.111111	0.100000	0.090909

In [309]:
```
df1.rdiv(1)
```

Out[309]:

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250000	0.200000	0.166667	0.142857
2	0.125000	0.111111	0.100000	0.090909

Table 5-5. Flexible arithmetic methods

Method	Description
`add, radd`	Methods for addition (+)
`sub, rsub`	Methods for subtraction (-)
`div, rdiv`	Methods for division (/)
`floordiv, rfloordiv`	Methods for floor division (//)
`mul, rmul`	Methods for multiplication (*)
`pow, rpow`	Methods for exponentiation (**)

Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value:

In [310]:
```
df1
```

Out[310]:

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

Processing math: 100%

In [311]:
```
df2
```

Out[311]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	NaN	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

In [312]:
```
df1.reindex(columns=df2.columns, fill_value=0)
```

Out[312]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	0
1	4.0	5.0	6.0	7.0	0
2	8.0	9.0	10.0	11.0	0

Operations between DataFrame and Series

In [313]:
```
arr = np.arange(12.).reshape((3, 4))
```

In [314]:
```
arr
```

Out[314]:
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

In [315]:
```
arr[0]
```

Out[315]:
```
array([0., 1., 2., 3.])
```

In [316]:
```
arr - arr[0]
```

Out[316]:
```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

In [317]:
```
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

In [318]:
```
series = frame.iloc[0]
```

Processing math: 100%

In [319]:
```
frame
```

Out[319]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [320]:
```
series
```

Out[320]:
```
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

In [321]:
```
frame - series
```

Out[321]:

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

If an index value is not found in either the DataFrame's columns or the Series's index, the objects will be reindexed to form the union:

In [325]:
```
series2 = pd.Series(range(3), index=['b', 'e', 'f'])
```

In [326]:
```
frame
```

Out[326]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [323]:
```
series2
```

Out[323]:
```
b    0
e    1
f    2
dtype: int64
```

Processing math: 100%

In [324]:
```
frame + series2
```

Out[324]:

	b	d	e	f
Utah	0.0	NaN	3.0	NaN
Ohio	3.0	NaN	6.0	NaN
Texas	6.0	NaN	9.0	NaN
Oregon	9.0	NaN	12.0	NaN

If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods. For example:

In [327]:
```
series3 = frame['d']
```

In [328]:
```
frame
```

Out[328]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [329]:
```
series3
```

Out[329]:
```
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64
```

In [330]:
```
frame.sub(series3, axis='index')
```

Out[330]:

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

Function Application and Mapping

NumPy ufuncs (element-wise array methods) also work with pandas objects:

Processing math: 100%

In [332]:
```
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

In [333]:
```
frame
```

Out[333]:

	b	d	e
Utah	-1.911782	1.414186	-0.045205
Ohio	-2.900189	0.714218	-0.160441
Texas	-0.039680	2.216894	1.189786
Oregon	0.957365	-0.200492	-0.892435

In [334]:
```
np.abs(frame)
```

Out[334]:

	b	d	e
Utah	1.911782	1.414186	0.045205
Ohio	2.900189	0.714218	0.160441
Texas	0.039680	2.216894	1.189786
Oregon	0.957365	0.200492	0.892435

Another frequent operation is applying a function on one-dimensional arrays to each column or row. DataFrame's **apply** method does exactly this:

In [335]:
```
f = lambda x: x.max() - x.min()
```

In [336]:
```
frame.apply(f)
```

Out[336]:
```
b    3.857554
d    2.417386
e    2.082221
dtype: float64
```

Here the function f, which computes the difference between the maximum and minimum of a Series, is invoked once on each column in frame. The result is a Series having the columns of frame as its index.

If you pass *axis='columns'* to apply, the function will be invoked once per row instead:

Processing math: 100%

In [339]:
```
frame.apply(f, axis='columns')
```

Out[339]:
```
Utah      3.325968
Ohio      3.614407
Texas     2.256574
Oregon    1.849800
dtype: float64
```

Many of the most common array statistics (like **sum** and **mean**) are DataFrame methods, so using apply is not necessary.

The function passed to apply need not return a scalar value; it can also return a Series with multiple values:

In [340]:
```
def f(x):
    return pd.Series([x.min(), x.max()], index=['min', 'max'])
```

In [341]:
```
frame.apply(f)
```

Out[341]:

	b	d	e
min	-2.900189	-0.200492	-0.892435
max	0.957365	2.216894	1.189786

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating-point value in frame. You can do this with **applymap**:

In [342]:
```
format = lambda x: '%.2f' % x
```

In [343]:
```
frame.applymap(format)
```

Out[343]:

	b	d	e
Utah	-1.91	1.41	-0.05
Ohio	-2.90	0.71	-0.16
Texas	-0.04	2.22	1.19
Oregon	0.96	-0.20	-0.89

The reason for the name **applymap** is that Series has a **map** method for applying an element-wise function:

Processing math: 100%

In [344]:
```
frame['e'].map(format)
```

Out[344]:
```
Utah      -0.05
Ohio      -0.16
Texas      1.19
Oregon    -0.89
Name: e, dtype: object
```

Sorting and Ranking

To sort lexicographically by row or column index, use the *sort*index_ method, which returns a new, sorted object:

In [345]:
```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
```

In [347]:
```
obj
```

Out[347]:
```
d    0
a    1
b    2
c    3
dtype: int64
```

In [348]:
```
obj.sort_index()
```

Out[348]:
```
a    1
b    2
c    3
d    0
dtype: int64
```

In [349]:
```
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
index=['three', 'one'],
columns=['d', 'a', 'b', 'c'])
```

In [350]:
```
frame.sort_index()
```

Out[350]:

	d	a	b	c
one	4	5	6	7
three	0	1	2	3

Processing math: 100%

In [351]:
```
frame.sort_index(axis=1)
```

Out[351]:

	a	b	c	d
three	1	2	3	0
one	5	6	7	4

In [352]:
```
frame.sort_index(axis=1, ascending=False)
```

Out[352]:

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

To sort a Series by its values, use its **sort_values** method:

In [354]:
```
obj = pd.Series([4, 7, -3, 2])
```

In [355]:
```
obj.sort_values()
```

Out[355]:
```
2   -3
3    2
0    4
1    7
dtype: int64
```

Any missing values are sorted to the end of the Series by default:

In [356]:
```
obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
```

In [357]:
```
obj.sort_values()
```

Out[357]:
```
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

When sorting a DataFrame, you can use the data in one or more columns as the sort keys. To do so, pass one or more column names to the **by** option of **sort_values**:

In [359]:
```
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
```

Processing math: 100%

In [360]:
```
frame
```

Out[360]:

	b	a
0	4	0
1	7	1
2	-3	0
3	2	1

In [361]:
```
frame.sort_values(by='b')
```

Out[361]:

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

In [362]:
```
frame.sort_values(by=['a', 'b'])
```

Out[362]:

	b	a
2	-3	0
0	4	0
3	2	1
1	7	1

In [363]:
```
frame.sort_values(by=['b', 'a'])
```

Out[363]:

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

Ranking assigns ranks from one through the number of valid data points in an array. The **rank** methods for Series and DataFrame are the place to look; by default rank breaks ties by assigning each group the mean rank (Equal values are assigned a rank that is the average of the ranks of those values):

In [370]:
```
obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
```

Processing math: 100%

In [371]:
```
obj.rank()
```

Out[371]:
```
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

Ranks can also be assigned according to the order in which they're observed in the data:

In [372]:
```
obj.rank(method='first')
```

Out[372]:
```
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

Here, instead of using the average rank 6.5 for the entries 0 and 2, they instead have been set to 6 and 7 because label 0 precedes label 2 in the data.

In [373]:
```
obj.rank(ascending=False, method='max')
```

Out[373]:
```
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

Table 5-6. Tie-breaking methods with rank

Method	Description
`'average'`	Default: assign the average rank to each entry in the equal group
`'min'`	Use the minimum rank for the whole group
`'max'`	Use the maximum rank for the whole group
`'first'`	Assign ranks in the order the values appear in the data
`'dense'`	Like `method='min'`, but ranks always increase by 1 in between groups rather than the number of equal elements in a group

Processing math: 100%

DataFrame can compute ranks over the rows or the columns:

In [376]:
```
frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],'c': [-2, 5, 8, -2
```

In [377]:
```
frame
```

Out[377]:

	b	a	c
0	4.3	0	-2.0
1	7.0	1	5.0
2	-3.0	0	8.0
3	2.0	1	-2.5

In [378]:
```
frame.rank()
```

Out[378]:

	b	a	c
0	3.0	1.5	2.0
1	4.0	3.5	3.0
2	1.0	1.5	4.0
3	2.0	3.5	1.0

In [380]:
```
frame.rank(axis="columns")
```

Out[380]:

	b	a	c
0	3.0	2.0	1.0
1	3.0	1.0	2.0
2	1.0	2.0	3.0
3	3.0	2.0	1.0

Axis Indexes with Duplicate Labels

While many pandas functions (like reindex) require that the labels be unique, it’s not mandatory. Let’s consider a small Series with duplicate indices:

In [381]:
```
obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
```

Processing math: 100%

In [382]:
```
obj
```

Out[382]:
```
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

The index's **is_unique** property can tell you whether its labels are unique or not:

In [383]:
```
obj.index.is_unique
```

Out[383]:
```
False
```

Data selection is one of the main things that behaves differently with duplicates. Indexing a label with multiple entries returns a Series, while single entries return a scalar value:

In [384]:
```
obj['a']
```

Out[384]:
```
a    0
a    1
dtype: int64
```

In [385]:
```
obj['c']
```

Out[385]:
```
4
```

The same logic extends to indexing rows in a DataFrame:

In [386]:
```
df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])
```

In [387]:
```
df
```

Out[387]:

	0	1	2
a	-0.952936	-0.429246	-0.400353
a	-0.818483	0.522370	-1.545153
b	-1.158105	-0.596782	-0.927029
b	0.244420	0.626243	1.946512

In [388]:
```
df.loc['b']
```

Out[388]:

	0	1	2
b	-1.158105	-0.596782	-0.927029
b	0.244420	0.626243	1.946512

Summarizing and Computing Descriptive Statistics

In [389]:
```
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
[np.nan, np.nan], [0.75, -1.3]],
index=['a', 'b', 'c', 'd'],
columns=['one', 'two'])
```

In [390]:
```
df
```

Out[390]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

Calling DataFrame's sum method returns a Series containing column sums:

In [391]:
```
df.sum()
```

Out[391]:
```
one     9.25
two    -5.80
dtype: float64
```

Passing **axis='columns'** or **axis=1** sums across the columns instead:

In [392]:
```
df.sum(axis='columns')
```

Out[392]:
```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled with the **skipna** option:

In [393]:
```
df.mean(axis='columns', skipna=False)
```

Out[393]:
```
a       NaN
b     1.300
c       NaN
d    -0.275
dtype: float64
```

List of common options for each reduction method:

- axis: Axis to reduce over; 0 for DataFrame's rows and 1 for columns
- skipna: Exclude missing values; True by default
- level: Reduce grouped by level if the axis is hierarchically indexed (MultiIndex)

Some methods, like **idxmin** and **idxmax**, return indirect statistics like the index value where the minimum or maximum values are attained:

In [394]:
```
df.idxmax()
```

Out[394]:
```
one    b
two    d
dtype: object
```

Other methods are accumulations:

In [395]:
```
df.cumsum()
```

Out[395]:

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

Another type of method is neither a reduction nor an accumulation. describe is one such example, producing multiple summary statistics in one shot:

Processing math: 100%

In [396]:
```
df.describe()
```

Out[396]:

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

On non-numeric data, **describe** produces alternative summary statistics:

In [400]:
```
obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
```

In [401]:
```
obj.describe()
```

Out[401]:
```
count     16
unique     3
top        a
freq       8
dtype: object
```

Processing math: 100%

Table 5-8. Descriptive and summary statistics

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index labels at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
prod	Product of all values
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (third moment) of values
kurt	Sample kurtosis (fourth moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute first arithmetic difference (useful for time series)
pct_change	Compute percent changes

Unique Values, Value Counts, and Membership

unique which gives you an array of the unique values in a Series:

In [407]:
```
obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

In [408]:
```
uniques = obj.unique()
```

In [409]:
```
uniques
```

Out[409]:
```
array(['c', 'a', 'd', 'b'], dtype=object)
```

value_counts computes a Series containing value frequencies:

Processing math: 100%

In [410]:
```
obj.value_counts()
```

Out[410]:
```
c    3
a    3
b    2
d    1
dtype: int64
```

value_counts is also available as a top-level pandas method that can be used with any array or sequence:

In [411]:
```
pd.value_counts(obj.values, sort=False)
```

Out[411]:
```
b    2
d    1
a    3
c    3
dtype: int64
```

isin performs a vectorized set membership check and can be useful in filtering a dataset down to a subset of values in a Series or column in a DataFrame:

In [412]:
```
obj
```

Out[412]:
```
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object
```

In [413]:
```
mask = obj.isin(['b', 'c'])
```

In [414]:
```
mask
```

Out[414]:
```
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

Processing math: 100%

In [415]:
```
obj[mask]
```

Out[415]:
```
0    c
5    b
6    b
7    c
8    c
dtype: object
```

Related to **isin** is the **Index.get_indexer** method, which gives you an index array from an array of possibly non-distinct values into another array of distinct values:

In [417]:
```
to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
```

In [418]:
```
unique_vals = pd.Series(['c', 'b', 'a'])
```

In [421]:
```
pd.Index(unique_vals).get_indexer(to_match)
```

Out[421]:
```
array([0, 2, 1, 1, 0, 2], dtype=int64)
```

Table 5-9. Unique, value counts, and set membership methods

Method	Description
isin	Compute boolean array indicating whether each Series value is contained in the passed sequence of values
match	Compute integer indices for each value in an array into another array of distinct values; helpful for data alignment and join-type operations
unique	Compute array of unique values in a Series, returned in the order observed
value_counts	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order

In some cases, you may want to compute a histogram on multiple related columns in a DataFrame. Here's an example:

In [425]:
```
data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
'Qu2': [2, 3, 1, 2, 3],
'Qu3': [1, 5, 2, 4, 4]})
```

Processing math: 100%

In [426]:
```
data
```

Out[426]:

	Qu1	Qu2	Qu3
0	1	2	1
1	3	3	5
2	4	1	2
3	3	2	4
4	4	3	4

Passing *pandas.value_counts* to this DataFrame's *apply* function gives:

In [427]:
```
result = data.apply(pd.value_counts).fillna(0)
```

In [428]:
```
result
```

Out[428]:

	Qu1	Qu2	Qu3
1	1.0	1.0	1.0
2	0.0	2.0	1.0
3	2.0	2.0	0.0
4	2.0	0.0	2.0
5	0.0	0.0	1.0

Here, the row labels in the result are the distinct values occurring in all of the col- umns. The values are the respective counts of these values in each column.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Processing math: 100%

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

Processing math: 100%