Fundamentals of Information Systems

Python Programming (for Data Science)

Master's Degree in Data Science

Gabriele Tolomei

gtolomei@math.unipd.it
University of Padua, Italy
2018/2019
November, 12 2018

Lecture 7: Introduction to `pandas`

What is `pandas`?

Ok, seriously what *is* `pandas`?

- Very likely, one of the most important tool for a (Pythonista) Data Scientist!
- It contains data structures and data manipulation tools designed to make data **cleaning** and **analysis** fast and easy in Python.
- Acronym derived from "**p**ython **an**d **d**ata **a**nalysi**s**" or, maybe, "**pan**el **da**ta".

What Does It Provide Us With?

- Often used in tandem with numerical computing tools like `numpy` and `scipy`, as well as machine learning modules like `statsmodels` and `scikit-learn`, and data visualization libraries like `matplotlib`.
- It uses significant parts of `numpy` style of array-based computing, especially vectorized functions for data processing without `for` loops.
- You can find more about `pandas` on the official website.

In [1]:

```python
"""
We import numpy as we will need it very often in combination with pandas
"""
import numpy as np
"""
As any other third-party module, the pandas module has to be imported before it can be used.
If you installed Python with Anaconda, pandas would be just available to you.
This is usually how pandas is imported and aliased. Although you could also
use another syntax like 'from pandas import *', I strongly encourage you to define an alias,
as this will help you to identify pandas' functions in your code.
"""
import pandas as pd
```

Introduction to `pandas` Data Structures

- Two workhorse data structures: `Series` and `DataFrame`.
- While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

`Series`

- A `Series` is a one-dimensional array-like object containing a **sequence of values** (similar to `numpy`'s arrays) and an associated array of data labels, called its `index`.
- The string representation of a `Series` displayed interactively shows the index on the left and the values on the right.
- If no index is specified for the data, a default one consisting of the integers 0 through $n-1$ (where n is the length of the data) is created.

In [2]:
```python
# Create a Series object (from a bulit-in Python list)
s = pd.Series([4, 7, -5, 3])
# Print the string representation of the Series object above
print(s)
print()
# We can get the array representation and index object of the Series
# via its values and index attributes, respectively
# Print the array of values
print("The values of the Series object are: {}".format(s.values))
# Print the array of indices
print("The index of the Series object is: {}".format(s.index)) # like range(4)
```

```
0    4
1    7
2   -5
3    3
dtype: int64

The values of the Series object are: [ 4  7 -5  3]
The index of the Series object is: RangeIndex(start=0, stop=4, step=1)
```

In [3]:
```
"""
Often it will be desirable to create a Series with an index
identifying each data point with a label.
"""
s2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
print(s2)
print()
print("The index of the Series object is: {}".format(s2.index))
```

```
d    4
b    7
a   -5
c    3
dtype: int64

The index of the Series object is: Index(['d', 'b', 'a', 'c'], dtype='object')
```

In [4]:
```python
"""
Compared with numpy arrays, you can use values in the index
when selecting single values or a set of values.
"""
# Accessing the element using the index value (i.e., not just an integer as with numpy!)
print("The value indexed by 'a' is: {}".format(s2['a']))
# Assign a new value corresponding to an index
s2['c'] = 42
print("After the assignment, the new Series object is:\n{}".format(s2))
print()
# Indexing a subset of the elements using a subset of the index
print("The sub-series is:\n{}".format(s2[['c', 'b', 'd']]))
```

```
The value indexed by 'a' is: -5
After the assignment, the new Series object is:
d     4
b     7
a    -5
c    42
dtype: int64

The sub-series is:
c    42
b     7
d     4
dtype: int64
```

In [5]:
```
"""
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.
"""
print("Select only the strictly positive elements of the series:\n{}".format(s2[s2 > 0]))
print()
double_s2 = s2 * 2
print("Double each element of the series:\n{}".format(double_s2))
print()
exp_s2 = np.exp(s2)
print("Apply the exponential function to each element of the series:\n{}".format(exp_s2))
# Note the implicit type conversion after the operation above
```

```
Select only the strictly positive elements of the series:
d     4
b     7
c    42
dtype: int64

Double each element of the series:
d     8
b    14
a   -10
c    84
dtype: int64

Apply the exponential function to each element of the series:
d    5.459815e+01
b    1.096633e+03
a    6.737947e-03
c    1.739275e+18
dtype: float64
```

In [6]:

```python
"""
Another way to think about a pandas Series is as a fixed-length, ordered built-in dict,
as it is a mapping of index values to data values.
Therefore, Series can be used in many contexts where you might use a dict.
"""
# Suppose you have a dictionary containing the following key-value pairs
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
# You can create a Series object from the dictionary above
s3 = pd.Series(sdata)
print("The series object created from the dictionary is:\n{}".format(s3))
```

```
The series object created from the dictionary is:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

In [7]:
```python
"""
When 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.
"""
states = ['California', 'Texas', 'Oregon', 'Ohio']
s4 = pd.Series(sdata, index=states)
print("The series object created from the dictionary and index is:\n{}".format(s4))
```

```
The series object created from the dictionary and index is:
California        NaN
Texas         71000.0
Oregon        16000.0
Ohio          35000.0
dtype: float64
```

In [8]:
```python
"""
Here, 3 values found in sdata were placed in the appropriate locations,
but since no value for 'California' was found, it appears as NaN (Not a Number),
which is considered in pandas to mark missing or NA values.
We will use the terms "missing" or "NA" to refer to missing data.
The 'isnull' and 'notnull' functions in pandas should be used to detect missing data.
"""
# Test for NA
# We can either invoke the method on the Series object
print("Check which elements of the series are null:\n{}".format(s4.isnull()))
# Or use the standard pandas function and pass the Series as input argument
#print("Check which elements of the series are null:\n{}".format(pd.isnull(s4)))
```

```
Check which elements of the series are null:
California     True
Texas         False
Oregon        False
Ohio          False
dtype: bool
```

In [9]:
```python
# Test for NOT NA
# We can either invoke the method on the Series object
print("Check which elements of the series are NOT null:\n{}".format(s4.notnull()))
# Or use the standard pandas function and pass the Series as input argument
#print("Check which elements of the series are NOT null:\n{}".format(pd.notnull(s4)))
```

```
Check which elements of the series are NOT null:
California    False
Texas          True
Oregon         True
Ohio           True
dtype: bool
```

In [10]:

```python
"""
Series automatically aligns differently-indexed data in arithmetic operations.
"""
print("Series s3:\n{}".format(s3))
print()
print("Series s4:\n{}".format(s4))
print()
print("Series s3 + s4:\n{}".format(s3 + s4))
```

```
Series s3:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64

Series s4:
California        NaN
Texas         71000.0
Oregon        16000.0
Ohio          35000.0
dtype: float64

Series s3 + s4:
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

In [11]:
```python
"""
Both the Series object itself and its index have a name attribute,
which integrates with other key areas of pandas functionality.
"""
s4.name = 'population'
s4.index.name = 'state'
print(s4)
```

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

In [12]:
```python
"""
A Series's index can be altered in place by assignment.
"""
s4.index = ['Alice', 'Bob', 'Carl', 'Denise']
print(s4)
# What happens if there is a length mismatch between the old and the new index?
```

```
Alice         NaN
Bob       71000.0
Carl      16000.0
Denise    35000.0
Name: population, dtype: float64
```

In [13]:

```
"""
A Series's index can be altered in place by assignment.
"""
s4.index = ['Alice', 'Bob', 'Carl', 'Denise', 'Eddie', 'Fran']
print(s4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-13-878932cea0c8> in <module>()
      2 A Series's index can be altered in place by assignment.
      3 """
----> 4 s4.index = ['Alice', 'Bob', 'Carl', 'Denise', 'Eddie', 'Fran']
      5 print(s4)

~/anaconda3/lib/python3.6/site-packages/pandas/core/generic.py in __setattr__(self, name,
 value)
   3625         try:
   3626             object.__getattribute__(self, name)
-> 3627             return object.__setattr__(self, name, value)
   3628         except AttributeError:
   3629             pass

pandas/_libs/properties.pyx in pandas._libs.properties.AxisProperty.__set__()

~/anaconda3/lib/python3.6/site-packages/pandas/core/series.py in _set_axis(self, axis, lab
els, fastpath)
    322         object.__setattr__(self, '_index', labels)
    323         if not fastpath:
--> 324             self._data.set_axis(axis, labels)
    325 
    326     def _set_subtyp(self, is_all_dates):

~/anaconda3/lib/python3.6/site-packages/pandas/core/internals.py in set_axis(self, axis, n
ew_labels)
   3072             raise ValueError('Length mismatch: Expected axis has %d elements, '
   3073                              'new values have %d elements' %
-> 3074                              (old_len, new_len))
   3075 
   3076         self.axes[axis] = new_labels

ValueError: Length mismatch: Expected axis has 4 elements, new values have 6 elements
```

`DataFrame`

- A `DataFrame` represents a rectangular table of data and contains an **ordered collection of columns**.
- Each column 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.
- Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays (the exact details are outside the scope of this class).

NOTE: While a `DataFrame` is physically two-dimensional, you can use it to represent higher-dimensional data in a tabular format using **hierarchical indexing**, one of the more advanced data-handling features in `pandas`.

In [14]:

```python
"""
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.
"""
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'population': [1.5, 1.7, 3.6, 2.4, 2.9]}

df = pd.DataFrame(data)
# The resulting DataFrame will have its index assigned automatically as with Series,
# and the columns are placed in sorted order
print("The DataFrame created is as follows:\n{}".format(df))
```

```
The DataFrame created is as follows:
   population   state  year
0         1.5    Ohio  2000
1         1.7    Ohio  2001
2         3.6    Ohio  2002
3         2.4  Nevada  2001
4         2.9  Nevada  2002
```

In [15]:
```python
"""
When you work with large DataFrames it is useful to show the first n rows
The 'head()' method without any input argument returns the first 5 rows.
Otherwise, you can call the method specifying the number of records to be displayed.
"""
print(df.head())
# You can also just type head()
df.head()
```

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

Out[15]:

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

In [16]:
```python
"""
If you specify a sequence of columns at creation time,
the DataFrame's columns will be arranged in that order.
"""
# Note that column names must reflect key names of the input dictionary 'data'
df2 = pd.DataFrame(data, columns=['year', 'state', 'population'])
print(df2)
```

```
   year   state  population
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
```

In [17]:

```python
"""
If you pass a column that isn't contained in the dict,
it will appear with missing values (NaN) in the result.
"""
# Here we are adding an extra column 'debt' and we assign a string-based index,
# instead of the default integer range.
df3 = pd.DataFrame(data, columns=['year', 'state', 'population', 'debt'],
                      index=['one', 'two', 'three', 'four', 'five'])
print(df3)
```

```
       year   state  population 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
```

In [18]:
```python
"""
A column in a DataFrame can be retrieved as a Series,
either by dict-like notation or by attribute.
"""
# Accessing column using dict-like notation
print(df3['state'])
print()
# Accessing column using attribute ('.column_name') notation
print(df3.population)

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

```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
Name: state, dtype: object

one      1.5
two      1.7
three    3.6
four     2.4
five     2.9
Name: population, dtype: float64
```

In [19]:

```python
"""
Rows can also be retrieved by position or name by a couple of methods,
such as the 'loc' indexing field (much more on this later).
"""
# Accessing the row indexed by 'three'
df3.loc['three']
```

Out[19]:

```
year          2002
state         Ohio
population     3.6
debt           NaN
Name: three, dtype: object
```

In [20]:
```
"""
Columns can be modified by assignment.
For example, the empty 'debt' column could be assigned a scalar value or an array of values.
"""
# Assigning the same scalar value to the column 'debt'
df3['debt'] = 73.42
print(df3)
```

```
       year   state  population   debt
one    2000    Ohio         1.5  73.42
two    2001    Ohio         1.7  73.42
three  2002    Ohio         3.6  73.42
four   2001  Nevada         2.4  73.42
five   2002  Nevada         2.9  73.42
```

In [21]:

```python
"""
Columns can be modified by assignment.
For example, the empty 'debt' column could be assigned a scalar value or an array of values.
"""
# Assigning an array of values to the column 'debt'
df3['debt'] = np.array([1.3, 6.4, 0.5, 0.0, -9.8])
print(df3)
# When assigning lists or arrays to a column,
# the value's length must match the length of the DataFrame.
```

```
       year   state  population  debt
one    2000    Ohio         1.5   1.3
two    2001    Ohio         1.7   6.4
three  2002    Ohio         3.6   0.5
four   2001  Nevada         2.4   0.0
five   2002  Nevada         2.9  -9.8
```

In [22]:

```python
"""
We can also assign a Series to a DataFrame column.
In this case, its labels will be realigned exactly to the DataFrame's index,
inserting missing values (NaN) in any holes.
"""
val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
df3['debt'] = val
print(df3)
# If one or more elements in the specified index are not present
# in the DataFrame's index, the assignment for those elements won't occur
```

```
       year   state  population  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
```

In [23]:
```python
"""
Assigning a column that doesn't exist will create a new column.
"""
# The effect of the following statement is to:
# 1. create a new column 'eastern'
# 2. assign to that column a value which results from the boolean expression
# Remember that boolean expressions computed over Series (or numpy arrays for that matters)
# return a Series (array)
df3['eastern'] = df3['state'] == 'Ohio'
print(df3)
```

```
       year   state  population  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
```

In [24]:
```python
"""
To delete a column, we can use the 'del' keyword as with a dict.
"""
del df3['eastern']
print(df3)
print()
print("The columns of the DataFrame are: {}".format(df3.columns))
```

```
       year   state  population  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

The columns of the DataFrame are: Index(['year', 'state', 'population', 'debt'], dtype='ob
ject')
```

Note: The column returned when indexing a `DataFrame` is a **view** on the underlying data, <u>not</u> a copy. Thus, any in-place modifications to the `Series` will be reflected in the `DataFrame`. The column can be explicitly copied using the `Series`'s `copy()` method.

In [25]:

```python
"""
Another possible way of creating a pandas DataFrame is by using
a dictionary of dictionaries (i.e., a nested dictionary).
With this approach, pandas will interpret the outer dictionary keys as the columns
and the inner keys as the row indices.
"""
population = {
    'Nevada': {2001: 2.4, 2002: 2.9},
    'California': {2000: 3.6, 2001: 3.7, 2002: 4.1},
    'Texas': {2000: 5.5, 2001: 5.8, 2002: 7.0},
}

df4 = pd.DataFrame(population)
print(df4)
```

```
      California  Nevada  Texas
2000         3.6     NaN    5.5
2001         3.7     2.4    5.8
2002         4.1     2.9    7.0
```

Possible Arguments 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.

Possible Arguments to `DataFrame()` Constructor

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 Masked-Array	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

`Index` Objects

- `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 used when constructing a `Series` or `DataFrame` is internally converted to an `Index`.
- They are ***immutable***, as this makes safer to share those objects among data structures.

In [26]:

```python
# Suppose we create a Series with an explicit index (i.e., a built-in list)
s = pd.Series(range(5), index=['a', 'b', 'c', 'd', 'e'])
# Store the index property of the Series object
index = s.index
print("The index associated with the series is: {}".format(index))
# As any sequence-like objects, Index can be sliced over
print("The sliced index associated with the series is: {}".format(index[1:4]))
```

```
The index associated with the series is: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
The sliced index associated with the series is: Index(['b', 'c', 'd'], dtype='object')
```

In [27]:
```python
# Index objects are IMMUTABLE
index[1] = 'beta'   # TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-67b49cf05c58> in <module>()
      1 # Index objects are IMMUTABLE
----> 2 index[1] = 'beta'   # TypeError

~/anaconda3/lib/python3.6/site-packages/pandas/core/indexes/base.py in __setitem__(self, k
ey, value)
   1722 
   1723     def __setitem__(self, key, value):
-> 1724         raise TypeError("Index does not support mutable operations")
   1725 
   1726     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

`Index` Object's Hierarchy

Class	Description
`Index`	The most general Index object, representing axis labels in a NumPy array of Python objects.
`Int64Index`	Specialized Index for integer values.
`Float64Index`	Specialized Index for floating point values.
`MultiIndex`	"Hierarchical" index object representing multiple levels of indexing on a single axis. Can be thought of as similar to an array of tuples.
`RangeIndex`	An integer index for the special case of a regularly spaced sequence, similar to the Python `range(start, stop, step)` function.
`CategoricalIndex`	An index of values with `category`
`DatetimeIndex`	Stores nanosecond timestamps (represented using NumPy's `datetime64` dtype).
`PeriodIndex`	Specialized Index for Period data (timespans).

`Index` Object's Methods and Properties

Method	Description
append	Concatenate with additional Index objects, producing a new Index
diff	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

In [28]:

```
"""
We can eliminate one or more entries from an axis using the 'drop' method.
This will return a new object with the entries that are left.
"""
s = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
print("Original Series:\n{}".format(s))
```

```
Original Series:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [29]:
```python
# Eliminate a single entry
new_s = s.drop('b')
print("Series after dropping entry 'b':\n{}".format(new_s))
print()
# Eliminate multiple entries
new_s = s.drop(['a', 'e'])
print("Series after dropping entries 'a' and 'e':\n{}".format(new_s))
```

```
Series after dropping entry 'b':
a    0.0
c    2.0
d    3.0
e    4.0
dtype: float64

Series after dropping entries 'a' and 'e':
b    1.0
c    2.0
d    3.0
dtype: float64
```

In [30]:
```
"""
With DataFrame, index values can be deleted from either axis: index (row) and column.
"""
df = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
print("Original DataFrame:\n{}".format(df))
```

```
Original DataFrame:
          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 [31]:

```python
# Eliminate entries on the default index axis (i.e., rows)
new_df = df.drop(['Utah', 'Ohio'])
print("Index-Dropped DataFrame (axis 0):\n{}".format(new_df))
print()
# Eliminate entries on the column axis
new_df2 = df.drop(['two', 'four'], axis=1)
print("Column-Dropped DataFrame (axis 1):\n{}".format(new_df2))
```

```
Index-Dropped DataFrame (axis 0):
          one  two  three  four
Colorado    4    5      6     7
New York   12   13     14    15

Column-Dropped DataFrame (axis 1):
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

In [32]:

```python
"""
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.
"""
# Eliminating a single entry on the row axis in-place
df.drop('New York', inplace=True)
print(df)
```

```
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
```

Indexing, Selecting, Slicing, and Filtering

- `Series` indexing works analogously to `numpy` array indexing.
- In addition to just integers, you can also use the `Series`'s index values.

In [33]:

```python
# Create a Series object
s = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
print(s)
print()
# Accessing the 2nd element using integer indexing
print("The 2nd element of the Series is: {}".format(s[1]))
# Accessing the 3rd element using index value 'c'
print("The 3rd element of the Series is: {}".format(s['c']))
```

```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64

The 2nd element of the Series is: 1.0
The 3rd element of the Series is: 2.0
```

In [34]:
```
# Selecting multiple entries using a list of integers
print("The 1st and 3rd elements of the Series are:\n{}"
      .format(s[[0, 2]]))
print()
# Selecting multiple entries using a list of index values
print("The elements corresponding to entry 'a' and 'c' of the Series are:\n{}"
      .format(s[['a', 'c']]))
```

```
The 1st and 3rd elements of the Series are:
a    0.0
c    2.0
dtype: float64

The elements corresponding to entry 'a' and 'c' of the Series are:
a    0.0
c    2.0
dtype: float64
```

In [35]:

```
# Slicing the series using integer slicing
print("The first two elements of the Series are:\n{}".format(s[:2]))
print()
# Slicing using labels behaves differently than normal Python integer slicing
# in that the endpoint is inclusive
print("The first two elements of the Series are:\n{}".format(s['a':'b']))
print()
# Filtering on a series according to a boolean condition
print("Select only those elements greater than 0 and less than or equal to 2:\n{}"
      .format(s[(s > 0) & (s <= 2)]))
```

```
The first two elements of the Series are:
a    0.0
b    1.0
dtype: float64

The first two elements of the Series are:
a    0.0
b    1.0
dtype: float64

Select only those elements greater than 0 and less than or equal to 2:
b    1.0
c    2.0
dtype: float64
```

In [36]:
```python
"""
Indexing into a DataFrame is for retrieving one or more columns,
either with a single value or sequence.
"""
df = pd.DataFrame(np.arange(16).reshape((4, 4)),
                  index=['Ohio', 'Colorado', 'Utah', 'New York'],
                  columns=['one', 'two', 'three', 'four'])
print(df)
```

```
          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 [37]:

```python
# Accessing a single column in a DataFrame using column values (axis 1)
print("Column 'two':\n{}".format(df['two']))
print()
# Accessing multiple columns in a DataFrame using column values (axis 1)
print("Column 'three' and 'one':\n{}".format(df[['three', 'one']]))
```

```
Column 'two':
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64

Column 'three' and 'one':
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

In [38]:
```python
"""
We can use integer slicing as well as boolean array to select rows.
"""
# Integer slicing inside of [] slices ONLY the rows
print("First two rows:\n{}".format(df[:2]))
print()
# Select all the rows corresponding to the entries where values in column 'three'
# are greater than 5 (this will filter out the first row)
print(df[df['three'] > 5])
print()
# Select all the rows corresponding to the entries where values in column 'two'
# are smaller than 7 (this will filter out the last two rows)
print(df[df['two'] < 7])
```

```
First two rows:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

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

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

In [39]:
```python
"""
Boolean indexing on DataFrame works similar to Series.
"""
print(df < 3)
```

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

In [40]:

```python
"""
Assignment over boolean indexing on a DataFrame
"""
df[df < 3] = 0
print(df)
# This makes DataFrame syntactically more like
# a two-dimensional numpy array in this particular case.
```

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

In [41]:
```python
"""
To indexing a DataFrame, there are two special operators: loc and iloc.
They 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).
"""
# Using 'loc' operator we can index both rows (axis 0) and columns (axis 1)
# by means of labels rather than integers.
# Extract the row corresponding to 'Utah'
print(df.loc['Utah'])
print()
# Using 'iloc' operator we can index both rows (axis 0) and columns (axis 1)
# by means of integers.
# Extract the 3rd row (which corresponds to 'Utah')
print("3rd row:\n{}".format(df.iloc[2]))
```

```
one       8
two       9
three    10
four     11
Name: Utah, dtype: int64

3rd row:
one       8
two       9
three    10
four     11
Name: Utah, dtype: int64
```

In [42]:

```
# Using 'loc' operator we can index both rows (axis 0) and columns (axis 1)
# by means of labels rather than integers.
# Extract the row corresponding to 'Utah' and the column corresponding to 'two'
print(df.loc['Utah', 'two'])
print()
# Using 'iloc' operator we can index both rows (axis 0) and columns (axis 1)
# by means of integers.
# Extract the 3rd row (which corresponds to 'Utah') and 2nd column ('two')
print("3rd row and 2nd column:\n{}".format(df.iloc[2,1]))
```

```
9

3rd row and 2nd column:
9
```

In [43]:

```python
# Using 'loc' operator we can index both rows (axis 0) and columns (axis 1)
# by means of labels rather than integers.
# Extract the row corresponding to 'Utah' and the columns 'two' and 'three'
print(df.loc['Utah', ['two', 'three']])
print()
# Using 'iloc' operator we can index both rows (axis 0) and columns (axis 1)
# by means of integers.
# Extract the 3rd row (Utah), 2nd and 3rd columns ('two', 'three')
print("3rd row; 2nd and 3rd columns:\n{}".format(df.iloc[2, [1,2]]))
```

```
two       9
three    10
Name: Utah, dtype: int64

3rd row; 2nd and 3rd columns:
two       9
three    10
Name: Utah, dtype: int64
```

In [44]:
```
print("1st and 4th row; 4th, 1st and 2nd column:\n{}"
      .format(df.loc[['Ohio','New York'], ['four', 'one', 'two']]))
print()
print("1st and 4th row; 4th, 1st and 2nd column:\n{}".format(df.iloc[[0, 3], [3, 0, 1]]))
```

```
1st and 4th row; 4th, 1st and 2nd column:
          four  one  two
Ohio         3    0    0
New York    15   12   13

1st and 4th row; 4th, 1st and 2nd column:
          four  one  two
Ohio         3    0    0
New York    15   12   13
```

In [45]:

```python
# Mixed label/integer selection (DEPRECATED!)
print("1st and 4th row; 4th, 1st and 2nd column:\n{}"
      .format(df.ix[['Ohio', 'New York'], [3, 0, 1]]))
```

```
1st and 4th row; 4th, 1st and 2nd column:
          four  one  two
Ohio         3    0    0
New York    15   12   13
```

```
/Users/gabriele/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:3: Deprecation
Warning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated
  This is separate from the ipykernel package so we can avoid doing imports until
```

In [46]:

```python
"""
Slicing over two axis of a DataFrame.
"""
# Using loc in combination with label slicing (rightmost extreme included!)
print(df.loc[:'Utah', 'two':'three'])
```

```
          two  three
Ohio        0      0
Colorado    5      6
Utah        9     10
```

In [47]:
```python
"""
Slicing over two axis of a DataFrame.
"""
# Using iloc in combination with integer slicing (rightmost extreme NOT included!)
print(df.iloc[1:, :3])
```

```
          one  two  three
Colorado    4    5      6
Utah        8    9     10
New York   12   13     14
```

In [48]:
```python
"""
Indexing and Slicing can be mixed (e.g., indexing on one axis and slicing on the other).
This can be done both using loc and iloc operator
"""
# Indexing over axis 0 and slicing over axis 1 using loc.
# row indexed by 'New York', column indexed by 'one', 'two' and 'three'
print(df.loc['New York', 'one':'three'])
print()
# Slicing over axis 0 and indexing over axis 1 using iloc.
# first two rows, 3rd column
print(df.iloc[:2, 2])
```

```
one      12
two      13
three    14
Name: New York, dtype: int64

Ohio        0
Colorado    6
Name: three, dtype: int64
```

Indexing Options for `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 of 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 label.
`df.iloc[:, where]`	Selects single column of 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).

Arithmetic and Data Alignment

- An important `pandas` feature is the behavior of arithmetic between objects with different indexes.
- When 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.
- The internal data alignment introduces `NaN` values in the label locations that don't overlap.

In [49]:

```
"""
Consider the following two Series objects.
"""
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
# Let's try to add s1 to s2 as follows.
print(s1 + s2)
```

```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

In [50]:

```
"""
In the case of DataFrame, alignment is performed on both the rows and the columns.
"""
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
                   index=['Ohio', 'Texas', 'Colorado'])
print(df1)
```

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

In [51]:
```
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                   index=['Utah', 'Ohio', 'Texas', 'Oregon'])
print(df2)
```

```
          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 [52]:
```python
"""
Adding these together returns a DataFrame whose index and columns
are the union of the ones in each DataFrame.
"""
print(df1 + df2)
```

```
            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
```

In [53]:

```
"""
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.
"""
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))
print(df1)
```

```
     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
```

```
In [54]: """
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.
"""
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))
print(df2)
```

```
      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 [55]:
```python
"""
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.
"""
# df1 + df2 will result in NaN on those unmatched indices
print(df1 + df2)
```

```
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
1   9.0  11.0  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 NaN
3   NaN   NaN   NaN   NaN NaN
```

In [56]:

```python
"""
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.
"""
# Using the add method on df1, we pass df2 and an argument to fill_value
print(df1.add(df2, fill_value=0))
```

```
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0  11.0  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
```

Arithmetic Operations

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 (**)

Operations between `DataFrame` and `Series`

- As with `numpy` arrays of different dimensions, arithmetic between `DataFrame` and `Series` is also defined.
- Let's first recall again how this works with `numpy`.

In [57]:

```python
# Let's define a two-dimensional numpy array (i.e., a 3-by-4 matrix)
X = np.arange(12).reshape(3,4)
print("X =\n{}".format(X))
# Let's define a one-dimensional numpy array (i.e., a 3-by-1 vector)
y = np.arange(3).reshape(3,1)
print("y =\n{}".format(y))
print()
print("X - y =\n{}".format(X-y))
```

```
X =
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
y =
[[0]
 [1]
 [2]]

X - y =
[[0 1 2 3]
 [3 4 5 6]
 [6 7 8 9]]
```

In [58]:

```python
"""
Let's see the same yet applied to a pandas DataFrame and a Series.
"""
df = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                  index=['Utah', 'Ohio', 'Texas', 'Oregon'])

print("DataFrame:\n{}".format(df))
print()
s = df.iloc[0]
print("Series:\n{}".format(s))
```

```
DataFrame:
          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

Series:
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

In [59]:
```
"""
By default, arithmetic between DataFrame and Series matches the index of the Series
on the DataFrame's columns, broadcasting down the rows.
"""
print(df - s)
```

```
          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
```

In [60]:

```
"""
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.
"""
s2 = pd.Series(range(3), index=['b', 'e', 'f'])
print(df + s2)
```

```
          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
```

In [61]:

```python
"""
If we want to instead broadcast over the columns, matching on the rows,
you have to use one of the arithmetic methods. For example:
"""
# Get a Series corresponding to the 'd' column
s3 = df['d']
print(s3)
# Subtract from the DataFrame the Series just extracted using axis
# The axis number is the axis to match on.
# In this case we mean to match on the DataFrame's row index (axis=0) and broadcast across.
diff_df = df.sub(s3, axis=0) # sub creates a new object
print(diff_df)
```

```
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64
          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

In [62]:
```
"""
numpy ufuncs (element-wise array methods) also work with pandas objects.
"""
df = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
                  index=['Utah', 'Ohio', 'Texas', 'Oregon'])
print(df)
print()
print(np.abs(df))
```

```
               b         d         e
Utah    0.423645  0.042218 -0.086047
Ohio   -0.177802  0.876929 -0.518464
Texas  -1.079930 -0.721962  0.197569
Oregon -0.078162 -1.255005  0.488826

               b         d         e
Utah    0.423645  0.042218  0.086047
Ohio    0.177802  0.876929  0.518464
Texas   1.079930  0.721962  0.197569
Oregon  0.078162  1.255005  0.488826
```

In [63]:

```python
"""
Another frequent operation is applying a function on 1-D arrays to each column or row.
DataFrame's 'apply' method does exactly this.
"""
# Define an anonymous (i.e., lambda) function
# Note that the argument x here is meant to be an array (not a scalar!)
f = lambda x: x.max() - x.min()
# Apply the lambda function column-wise (default)
print("Column-wise application of lambda:\n{}".format(df.apply(f)))
# equals to df.apply(f, axis=0)
print()
# Apply the lambda function row-wise
print("Row-wise application of lambda:\n{}".format(df.apply(f, axis=1)))
```

```
Column-wise application of lambda:
b    1.503575
d    2.131934
e    1.007289
dtype: float64

Row-wise application of lambda:
Utah      0.509693
Ohio      1.395392
Texas     1.277499
Oregon    1.743831
dtype: float64
```

`apply`

- The function passed to `apply` is expected to work on a one-dimensional array of data (i.e., a `Series`) as input.
- It **does not** need to necessarily return a scalar value as output, in fact it can also return a `Series` with multiple values.
- Many of the most common array statistics (like `sum` and `mean`) are `DataFrame` built-in methods, therefore using `apply` is **not** necessary.

In [64]:
```python
"""
Consider the following function which returns a Series instead of a scalar value.
"""
def foo(x):
    return pd.Series([x.min(), x.max()], index=['min', 'max'])

print(df.apply(foo))
```

```
            b         d         e
min -1.079930 -1.255005 -0.518464
max  0.423645  0.876929  0.488826
```

`applymap`

- Sometimes, we may want to apply a function to each element of the `DataFrame`.
- In other words, the input of the function to be applied is a scalar rather than a one-dimensional array.
- `applymap` is the function to be used!

In [65]:
```python
"""
For example, suppose we want to apply a string format to each element of a DataFrame.
"""
# define the function to be applied
str_format = lambda x: '%.2f' % x

print(df.applymap(str_format))
```

```
            b      d      e
Utah     0.42   0.04  -0.09
Ohio    -0.18   0.88  -0.52
Texas   -1.08  -0.72   0.20
Oregon  -0.08  -1.26   0.49
```

`map`

- The `applymap` function above is implemented through the `map` function available for `Series` objects.
- In other words, `map` is the `Series`' corresponding of `DataFrame`'s `applymap`.

In [66]:
```
# Now the function `str_format` is passed as input to Series' `map` function
df['e'].map(str_format)
```

Out[66]:
```
Utah      -0.09
Ohio      -0.52
Texas      0.20
Oregon     0.49
Name: e, dtype: object
```

`apply`, `applymap`, and `map`

Summing up, `apply` works on a row/column basis of a `DataFrame`, `applymap` works element-wise on a `DataFrame`, and `map` works element-wise on a `Series`.

Sorting by Index Using `sort_index`

- To sort lexicographically by row or column index, use the `sort_index` method, which returns a **new**, sorted object (i.e., not *in-place* sorting).
- `sort_index` works both for `Series` and `DataFrame` objects, with the latter we can sort by index on either axis.
- By default, data is sorted in **ascending order**, but can be sorted in descending order as well by calling the `sort_index` method with `ascending=False` argument.

In [67]:
```python
"""
Usage of sort_index in Series objects.
"""
# Define a Series object
s = pd.Series(range(4), index=['d', 'a', 'c', 'b'])
print("Original Series:\n{}".format(s))
print()
print("Sort Series by index:\n{}".format(s.sort_index()))
```

```
Original Series:
d    0
a    1
c    2
b    3
dtype: int64

Sort Series by index:
a    1
b    3
c    2
d    0
dtype: int64
```

In [68]:
```python
"""
Using sort_index with DataFrame objects, we can specify which axis the index refers to.
"""
# Create a DataFrame object
df = pd.DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
                     columns=['d', 'a', 'b', 'c'])
print("Original DataFrame:\n{}".format(df))
print()
# By default, sort_index operates on axis 0 (i.e., rows)
print("Sort DataFrame by row index:\n{}".format(df.sort_index()))
```

```
Original DataFrame:
       d  a  b  c
three  0  1  2  3
one    4  5  6  7

Sort DataFrame by row index:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3
```

In [69]:

```
print("Original DataFrame:\n{}".format(df))
print()
# sort_index can also operate on axis 1 (i.e., columns)
print("Sort DataFrame by column index:\n{}".format(df.sort_index(axis=1)))
```

```
Original DataFrame:
       d  a  b  c
three  0  1  2  3
one    4  5  6  7

Sort DataFrame by column index:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

Sorting by Values Using `sort_values`

- Both `Series` and `DataFrame` objects can be sorted by **values** using the `sort_values` method.
- Sorting the values of a `Series` behaves as expected; by convention, any missing values (`NaN`) are placed at the end of the sorted object.
- When sorting a `DataFrame`, we can use one or more columns as the sort keys. To do so, we need to pass one or more column names as input arguments of `sort_values`.
- Again, sorting happens to be in **ascending order** by default but this can be changed using the `ascending=False` argument.

In [70]:
```
"""
Apply sort_values to a Seres object.
"""
# Create a Series object
s = pd.Series([4, 7, -3, 2])
print("Original Series:\n{}".format(s))
print()
print("Sort Series by values:\n{}".format(s.sort_values()))
```

```
Original Series:
0    4
1    7
2   -3
3    2
dtype: int64

Sort Series by values:
2   -3
3    2
0    4
1    7
dtype: int64
```

In [71]:
```
"""
Apply sort_values to a Seres object which contains missing values.
"""
# Create a Series object with missing values
s = pd.Series([4, np.nan, 5, -1, np.nan, 2])
print("Original Series:\n{}".format(s))
print()
print("Sort Series by values:\n{}".format(s.sort_values()))
```

```
Original Series:
0    4.0
1    NaN
2    5.0
3   -1.0
4    NaN
5    2.0
dtype: float64

Sort Series by values:
3   -1.0
5    2.0
0    4.0
2    5.0
1    NaN
4    NaN
dtype: float64
```

In [72]:

```python
"""
Apply sort_values to a DataFrame object.
"""
# Create a DataFrame object
df = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
print("Original DataFrame:\n{}".format(df))
print()
# Sort values by a single column
print("Sort DataFrame by values contained in column 'b':\n{}"
      .format(df.sort_values(by='b')))
```

```
Original DataFrame:
   a  b
0  0  4
1  1  7
2  0 -3
3  1  2

Sort DataFrame by values contained in column 'b':
   a  b
2  0 -3
3  1  2
0  0  4
1  1  7
```

In [73]:

```
# Sort values by multiple columns
print("Sort DataFrame by values contained in 2 columns 'a' and 'b':\n{}"
      .format(df.sort_values(by=['a','b'])))
print()
print("Sort DataFrame by values contained in 2 columns 'a' and 'b' (not ascending):\n{}"
      .format(df.sort_values(by=['a','b'], ascending=False)))
```

```
Sort DataFrame by values contained in 2 columns 'a' and 'b':
   a  b
2  0 -3
0  0  4
3  1  2
1  1  7

Sort DataFrame by values contained in 2 columns 'a' and 'b' (not ascending):
   a  b
1  1  7
3  1  2
0  0  4
2  0 -3
```

Summarizing and Computing Descriptive Statistics

- `pandas` objects are equipped with a set of common mathematical and statistical methods.
- Most of these fall into the category of **reductions** or **summary statistics**
- Methods that extract a single value (like the **sum** or **mean**) from a `Series` or a `Series` of values from the rows or columns of a `DataFrame`.
- Compared with the similar methods of `numpy` arrays, they have built-in handling for missing data.

In [74]:

```python
# Consider a DataFrame object as follows
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'])
print(df)
```

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

In [75]:

```python
# By default, aggregation function operates on axis 0 (i.e., rows) as usual
# For example, calling the sum method returns a Series containing column-wise sums.
print("Column-wise sum:\n{}".format(df.sum()))
print()
# Instead, if we want row-wise sum, we can of course specify axis=1.
print("Row-wise sum:\n{}".format(df.sum(axis=1)))
```

```
Column-wise sum:
one    9.25
two   -5.80
dtype: float64

Row-wise sum:
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

In [76]:

```python
# Note that NaN values are excluded unless the entire slice (row or column) is NaN.
# This can be disabled using the skipna option.
print("Row-wise Average including NaN:\n{}".format(df.mean(axis=1, skipna=False)))
```

```
Row-wise Average including NaN:
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

In [77]:

```python
"""
A useful method is 'describe' which provides multiple summary statistics in one shot.
"""
print(df.describe())
```

```
            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
```

In [78]:

```
"""
If the Series or DataFrame object does not contain numeric data
other statistics are produced.
"""
# Let's create a Series object containing non-numeric data
s = pd.Series(['a', 'a', 'b', 'c'] * 4)
print("Original Series:\n{}".format(s))
```

```
Original Series:
0     a
1     a
2     b
3     c
4     a
5     a
6     b
7     c
8     a
9     a
10    b
11    c
12    a
13    a
14    b
15    c
dtype: object
```

In [79]:
```python
# Now, let's call the describe method
print(s.describe())
```

```
count     16
unique     3
top        a
freq       8
dtype: object
```

Table of Descriptive and Summary Statistics (1 of 2)

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 values 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

Table of Descriptive and Summary Statistics (2 of 2)

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 (3rd moment) of values
kurt	Sample kurtosis (4th moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st arithmetic difference (useful for time series)
pct_change	Compute percent changes

Unique Values, Value Counts, and Membership

- `unique` is a function that gives you an array of the unique values in a `Series`.
- The unique values are not necessarily returned in sorted order, but could be sorted afterwards (e.g., by calling `np.sort()`).
- `value_counts` computes a `Series` containing value frequencies.
- `isin` performs a vectorized set membership check.

In [80]:
```
"""
Usage of unique method.
"""
# Create a Series object
s = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
# Get unique values and store them in a variable
uniques = s.unique()
print("Unique values are: {}".format(uniques))
print("Unique sorted values are: {}".format(np.sort(uniques)))
```

```
Unique values are: ['c' 'a' 'd' 'b']
Unique sorted values are: ['a' 'b' 'c' 'd']
```

In [81]:

```python
"""
Usage of value_counts method.
"""
# Either as a method
print("Value counts:\n{}".format(s.value_counts(sort=False)))
print()
# Or as a built-in pandas function
print("Value counts:\n{}".format(pd.value_counts(s, sort=False)))
```

```
Value counts:
a    3
d    1
b    2
c    3
dtype: int64

Value counts:
a    3
d    1
b    2
c    3
dtype: int64
```

In [82]:
```
"""
Usage of isin method.
"""
# Create a boolean mask
mask = s.isin(['b', 'c'])
print("Boolean mask:\n{}".format(mask))
print()
print("Accessing elements of the Series object using the mask:\n{}".format(s[mask]))
```

```
Boolean mask:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool

Accessing elements of the Series object using the mask:
0    c
5    b
6    b
7    c
8    c
dtype: object
```

In [83]:

```python
"""
Many times, it will be useful to compute a histogram on multiple columns in a DataFrame.
"""
data = pd.DataFrame({'Q1': [1, 3, 4, 3, 4],
                     'Q2': [2, 3, 1, 2, 3],
                     'Q3': [1, 5, 2, 4, 4]})

print("Original dataset:\n{}".format(data))
print()
# Passing pandas.value_counts to this DataFrame's apply function gives the following
freq = data.apply(pd.value_counts).fillna(0)
print("Column-wise frequency table:\n{}".format(freq))
```

```
Original dataset:
   Q1  Q2  Q3
0   1   2   1
1   3   3   5
2   4   1   2
3   3   2   4
4   4   3   4

Column-wise frequency table:
    Q1   Q2   Q3
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
```

In [84]:

```python
# If we specify axis=1 the frequency is computed row-wise
freq = data.apply(pd.value_counts, axis=1).fillna(0)
print("Row-wise frequency table:\n{}".format(freq))
```

```
Row-wise frequency table:
     1    2    3    4    5
0  2.0  1.0  0.0  0.0  0.0
1  0.0  0.0  2.0  0.0  1.0
2  1.0  1.0  0.0  1.0  0.0
3  0.0  1.0  1.0  1.0  0.0
4  0.0  0.0  1.0  2.0  0.0
```

Table of 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.