Analysis of unstructered data

Lecture 2 - Introduction to pandas module

Janusz Szwabiński

References:

* homepage of the Pandas project: http://pandas.pydata.org/ (http://pandas.pydata.org/)

Pandas - Python Data Analysis Library

* fast and flexible data structures - tables with named rows and columns
* real world data analysis in Python
* the ultimate goal - to become **the most powerful and flexible open source data analysis / manipulation tool available in any language**
* important features:
  * easy handling of missing data
  * size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects
  * automatic and explicit data alignment
  * flexible group by functionality to perform split-apply-combine operations on data sets
  * intelligent label-based slicing, fancy indexing, and subsetting of large data sets
  * intuitive merging and joining data sets
  * flexible reshaping and pivoting of data sets
  * hierarchical labeling of axes
  * robust IO tools for loading data from flat files (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast HDF5 format
  * time series-specific functionality: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging, etc

Data structures at a glance

Dimensions	Name	Description
1	Series	1D labeled homogeneously-typed array
2	DataFrame	General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed columns
3	Panel	General 3D labeled, also size-mutable array (**deprecated!!!**)

- all pandas data structures are value-mutable (the values they contain can be altered) but not always size-mutable
  - the length of a Series cannot be changed
  - columns can be inserted into a DataFrame
- the vast majority of methods produce new objects and leave the input data untouched $\rightarrow$ **immutability** is favored where sensible

Series

In [2]:

```
import numpy as np
import pandas as pd
```

- a one-dimensional labeled array
- homogeneously-typed
- capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.)
- axis labels are collectively referred to as the index.
- basic method to create a Series:

```
s = pd.Series(data,index=index_list)
```

- `index` - list of labels
- `data`:
  - a Python dict
  - an ndarray
  - a scalar value (like 5)

Creating a Series from ndarray

- `index` must be the same length as data
- if no `index` passed, one will be created having values [0, ..., len(data) - 1]

In [3]:

```
s = pd.Series(np.random.randn(5))
```

In [4]:

```
s
```

Out[4]:

```
0    1.031707
1   -0.549604
2    1.363238
3    0.949471
4   -1.390551
dtype: float64
```

In [5]:

```
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e']) #labels provi
ded by the user
```

In [6]:

```
s
```

Out[6]:

```
a    0.665696
b    1.077520
c   -1.342475
d   -0.383903
e   -0.122880
dtype: float64
```

In [7]:

```
s.index
```

Out[7]:

```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

Creating a Series from a dict

- if index is passed, the values in data corresponding to the labels in the index will be pulled out
- otherwise, an index will be constructed from the sorted keys of the dict (if possible)

In [9]:

```
d = {'a' : 0., 'b' : 1., 'c' : 2.}
pd.Series(d)
```

Out[9]:

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

In [12]:

```
pd.Series(d, index=['b', 'c', 'd', 'a'])
```

Out[12]:

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

NaN (*not a number*) is the standard missing data marker used in pandas.

Creating a Series from scalar value

- an index must be provided
- the value will be repeated to match the length of index

In [9]:

```
pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
```

Out[9]:

```
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Data manipulation

Series acts similarly to an ndarray:

In [13]:

```
print(s)
s[0]
```

```
a    0.665696
b    1.077520
c   -1.342475
d   -0.383903
e   -0.122880
dtype: float64
```

Out[13]:

```
0.66569586843579054
```

In [14]:

```
s[:3]
```

Out[14]:

```
a    0.665696
b    1.077520
c   -1.342475
dtype: float64
```

In [15]:

```
s[s > s.median()] #elements greater than median
```

Out[15]:

```
a    0.665696
b    1.077520
dtype: float64
```

In [16]:

```
s > s.median()
```

Out[16]:

```
a     True
b     True
c    False
d    False
e    False
dtype: bool
```

In [17]:

```
s[[4, 3, 1]]
```

Out[17]:

```
e   -0.122880
d   -0.383903
b    1.077520
dtype: float64
```

In [18]:

```
np.exp(s)
```

Out[18]:

```
a    1.945844
b    2.937386
c    0.261198
d    0.681198
e    0.884370
dtype: float64
```

It is also similar to a fixed-size dict:

In [19]:

```
s['a']
```

Out[19]:

```
0.66569586843579054
```

In [20]:

```
'e' in s
```

Out[20]:

```
True
```

In [21]:

```
'f' in s
```

Out[21]:

```
False
```

In [22]:

```
s['f']
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/usr/local/lib/python3.5/dist-packages/pandas/indexes/base.py in get_value(self, series, key)
   2174             try:
-> 2175                 return tslib.get_value_box(s, key)
   2176             except IndexError:

pandas/tslib.pyx in pandas.tslib.get_value_box (pandas/tslib.c:18246)()

pandas/tslib.pyx in pandas.tslib.get_value_box (pandas/tslib.c:17880)()

TypeError: 'str' object cannot be interpreted as an integer

During handling of the above exception, another exception occurred:

KeyError                                  Traceback (most recent call last)
<ipython-input-22-f7a405991146> in <module>()
----> 1 s['f']

/usr/local/lib/python3.5/dist-packages/pandas/core/series.py in __getitem__(self, key)
    599         key = com._apply_if_callable(key, self)
    600         try:
--> 601             result = self.index.get_value(self, key)
    602 
    603             if not is_scalar(result):

/usr/local/lib/python3.5/dist-packages/pandas/indexes/base.py in get_value(self, series, key)
   2181                     raise InvalidIndexError(key)
   2182                 else:
-> 2183                     raise e1
   2184             except Exception:  # pragma: no cover
   2185                 raise e1

/usr/local/lib/python3.5/dist-packages/pandas/indexes/base.py in get_value(self, series, key)
   2167         try:
   2168             return self._engine.get_value(s, k,
-> 2169                                           tz=getattr(series.dtype, 'tz', None))
   2170         except KeyError as e1:
   2171             if len(self) > 0 and self.inferred_type in ['integer', 'boolean']:

pandas/index.pyx in pandas.index.IndexEngine.get_value (pandas/index.c:3567)()

pandas/index.pyx in pandas.index.IndexEngine.get_value (pandas/index.c:3250)()

pandas/index.pyx in pandas.index.IndexEngine.get_loc (pandas/index.c:4289)()

pandas/src/hashtable_class_helper.pxi in pandas.hashtable.PyObjectHa
```

```
shTable.get_item (pandas/hashtable.c:13733)()

pandas/src/hashtable_class_helper.pxi in pandas.hashtable.PyObjectHa
shTable.get_item (pandas/hashtable.c:13687)()

KeyError: 'f'
```

Using the get method, a missing label will return `None` or specified default:

In [24]:

```
print(s.get('f'))
```

```
None
```

In [25]:

```
s.get('f', np.nan)
```

Out[25]:

```
nan
```

As with raw NumPy arrays, looping through Series value-by-value is usually not necessary:

In [26]:

```
s + s
```

Out[26]:

```
a    1.331392
b    2.155040
c   -2.684950
d   -0.767805
e   -0.245760
dtype: float64
```

In [27]:

```
s * 2
```

Out[27]:

```
a    1.331392
b    2.155040
c   -2.684950
d   -0.767805
e   -0.245760
dtype: float64
```

Series can be passed into most NumPy methods expecting an ndarray:

In [29]:

```
np.exp(s)
```

Out[29]:

```
a    1.945844
b    2.937386
c    0.261198
d    0.681198
e    0.884370
dtype: float64
```

A key difference between Series and ndarray is that operations between Series automatically align the data based on label (no need to consider whether the Series involved in a computation have the same labels):

In [30]:

```
s[1:]
```

Out[30]:

```
b    1.077520
c   -1.342475
d   -0.383903
e   -0.122880
dtype: float64
```

In [31]:

```
s[:-1]
```

Out[31]:

```
a    0.665696
b    1.077520
c   -1.342475
d   -0.383903
dtype: float64
```

In [32]:

```
s[1:] + s[:-1]
```

Out[32]:

```
a         NaN
b    2.155040
c   -2.684950
d   -0.767805
e         NaN
dtype: float64
```

- the result of an operation between unaligned Series will have the **union** of the indexes involved
- if a label is not found in one Series or the other, the result will be marked as missing NaN

DataFrame

- a 2-dimensional labeled data structure with columns of potentially different types
- may be interpreted as a spreadsheet or SQL table, or a dict of Series objects
- the most commonly used pandas object
- accepts many different kinds of input:
  - dict of 1D ndarrays, lists, dicts, or Series
  - 2-D numpy.ndarray
  - structured or record ndarray
  - a Series
  - another DataFrame
- along with the data, you can optionally pass index (row labels) and columns (column labels) arguments
- if axis labels are not passed, they will be constructed from the input data based on **common sense rules**

Creating DataFrame from a dict

In [33]:

```
d = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
     'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
```

In [34]:

```
df = pd.DataFrame(d)
```

In [35]:

```
df
```

Out[35]:

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

In [36]:

```
pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
```

Out[36]:

	two	three
d	4.0	NaN
b	2.0	NaN
a	1.0	NaN

In [37]:

```
df.index #row labels
```

Out[37]:

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

In [38]:

```
df.columns #column labels
```

Out[38]:

```
Index(['one', 'two'], dtype='object')
```

Creating DataFrame from a dict of lists or ndarrays

- ndarrays must all be the same length

In [39]:

```
d = {'one' : [1., 2., 3., 4.],
     'two' : [4., 3., 2., 1.]}
```

In [40]:

```
pd.DataFrame(d)
```

Out[40]:

	one	two
0	1.0	4.0
1	2.0	3.0
2	3.0	2.0
3	4.0	1.0

In [41]:

```
d1 = {'one' : [1., 2., 3., 4.],
      'two' : [4., 3., 2.]}
```

In [42]:

```
pd.DataFrame(d1)
```

```
---------------------------------------------------------------------
-------
ValueError                                Traceback (most recent cal
l last)
<ipython-input-42-7a08febeebf8> in <module>()
----> 1 pd.DataFrame(d1)

/usr/local/lib/python3.5/dist-packages/pandas/core/frame.py in __ini
t__(self, data, index, columns, dtype, copy)
    264                                  dtype=dtype, copy=copy)
    265         elif isinstance(data, dict):
--> 266             mgr = self._init_dict(data, index, columns, dtyp
e=dtype)
    267         elif isinstance(data, ma.MaskedArray):
    268             import numpy.ma.mrecords as mrecords

/usr/local/lib/python3.5/dist-packages/pandas/core/frame.py in _init
_dict(self, data, index, columns, dtype)
    400             arrays = [data[k] for k in keys]
    401 
--> 402         return _arrays_to_mgr(arrays, data_names, index, col
umns, dtype=dtype)
    403 
    404     def _init_ndarray(self, values, index, columns, dtype=No
ne, copy=False):

/usr/local/lib/python3.5/dist-packages/pandas/core/frame.py in _arra
ys_to_mgr(arrays, arr_names, index, columns, dtype)
   5407     # figure out the index, if necessary
   5408     if index is None:
-> 5409         index = extract_index(arrays)
   5410     else:
   5411         index = _ensure_index(index)

/usr/local/lib/python3.5/dist-packages/pandas/core/frame.py in extra
ct_index(data)
   5455             lengths = list(set(raw_lengths))
   5456             if len(lengths) > 1:
-> 5457                 raise ValueError('arrays must all be same le
ngth')
   5458 
   5459             if have_dicts:

ValueError: arrays must all be same length
```

Creating DataFrame from structured or record array

In [45]:

```
data = np.zeros((2,), dtype=[('A', 'i4'),('B', 'f4'),('C', 'a10')]) #columns of
 different types
```

In [46]:

```
data
```

Out[46]:

```
array([(0,  0., b''), (0,  0., b'')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])
```

In [47]:

```
data[:] = [(1,2.,'Hello'), (2,3.,"World")]
```

In [48]:

```
pd.DataFrame(data)
```

Out[48]:

	A	B	C
0	1	2.0	b'Hello'
1	2	3.0	b'World'

In [49]:

```
pd.DataFrame(data, index=['first', 'second'])
```

Out[49]:

	A	B	C
first	1	2.0	b'Hello'
second	2	3.0	b'World'

In [50]:

```
pd.DataFrame(data, columns=['C', 'A', 'B'])
```

Out[50]:

	C	A	B
0	b'Hello'	1	2.0
1	b'World'	2	3.0

Creating DataFrame from list of dicts

In [51]:

```
data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
pd.DataFrame(data2)
```

Out[51]:

	a	b	c
0	1	2	NaN
1	5	10	20.0

In [52]:

```
pd.DataFrame(data2, index=['first', 'second'])
```

Out[52]:

	a	b	c
first	1	2	NaN
second	5	10	20.0

In [53]:

```
pd.DataFrame(data2, columns=['a', 'b'])
```

Out[53]:

	a	b
0	1	2
1	5	10

Creating DataFrame from a dict of tuples

- automatical creation of a multi-indexed frame possible:

In [54]:

```
pd.DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2},
              ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},
              ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},
              ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8},
              ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}})
```

Out[54]:

		a			b	
		a	**b**	**c**	**a**	**b**
A	**B**	4.0	1.0	5.0	8.0	10.0
	C	3.0	2.0	6.0	7.0	NaN
	D	NaN	NaN	NaN	NaN	9.0

Creating DataFrame from a Series

- the result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (if provided)

In [50]:

```
s
```

Out[50]:

```
a   -0.968054
b    1.059388
c    0.337526
d    1.549708
e    2.131306
dtype: float64
```

In [51]:

```
pd.DataFrame(s)
```

Out[51]:

	0
a	-0.968054
b	1.059388
c	0.337526
d	1.549708
e	2.131306

In [52]:

```
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'], name="losowan
ie")
```

In [53]:

```
pd.DataFrame(s) #series name becomes column label
```

Out[53]:

	losowanie
a	0.127984
b	-0.559549
c	-0.931808
d	-0.057773
e	-1.710059

Column operations

- DataFrame can be treated as a dict of like-indexed Series objects
- getting, setting, and deleting columns works with the same syntax as the analogous dict operations

In [55]:

```
df
```

Out[55]:

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

In [56]:

```
df['one']
```

Out[56]:

```
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

In [57]:

```
df['three'] = df['one'] * df['two']
```

In [58]:

```
df['flag'] = df['one'] > 2
```

In [59]:

```
df
```

Out[59]:

	one	two	three	flag
a	1.0	1.0	1.0	False
b	2.0	2.0	4.0	False
c	3.0	3.0	9.0	True
d	NaN	4.0	NaN	False

In [60]:

```
del df['two']
df
```

Out[60]:

	one	three	flag
a	1.0	1.0	False
b	2.0	4.0	False
c	3.0	9.0	True
d	NaN	NaN	False

In [61]:

```
three = df.pop('three')
```

In [62]:

```
three
```

Out[62]:

```
a    1.0
b    4.0
c    9.0
d    NaN
Name: three, dtype: float64
```

When inserting a scalar value, it will be propagated to fill the column:

In [63]:

```
df['foo'] = 'bar'
```

In [64]:

```
df
```

Out[64]:

	one	flag	foo
a	1.0	False	bar
b	2.0	False	bar
c	3.0	True	bar
d	NaN	False	bar

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame’s index:

In [65]:

```
df['one_trunc'] = df['one'][:2]
```

In [66]:

```
df
```

Out[66]:

	one	flag	foo	one_trunc
a	1.0	False	bar	1.0
b	2.0	False	bar	2.0
c	3.0	True	bar	NaN
d	NaN	False	bar	NaN

- by default columns get inserted at the end
- with the `insert` function we can choose the location for a new column:

In [67]:

```
df.insert(1, 'bar', df['one'])
df
```

Out[67]:

	one	bar	flag	foo	one_trunc
a	1.0	1.0	False	bar	1.0
b	2.0	2.0	False	bar	2.0
c	3.0	3.0	True	bar	NaN
d	NaN	NaN	False	bar	NaN

Indexing/selection

Operation	Syntax	Result
Select column	`df[col]`	Series
Select row by label	`df.loc[label]`	Series
Select row by integer location	`df.iloc[loc]`	Series
Slice rows	`df[5:10]`	DataFrame
Slice rows by boolean vector	`df[bool_vec]`	DataFrame

In [68]:

```
df
```

Out[68]:

	one	bar	flag	foo	one_trunc
a	1.0	1.0	False	bar	1.0
b	2.0	2.0	False	bar	2.0
c	3.0	3.0	True	bar	NaN
d	NaN	NaN	False	bar	NaN

In [69]:

```
df.loc[1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-69-a3732eea6ed6> in <module>()
----> 1 df.loc[1]

/usr/local/lib/python3.5/dist-packages/pandas/core/indexing.py in __getitem__(self, key)
   1309             return self._getitem_tuple(key)
   1310         else:
-> 1311             return self._getitem_axis(key, axis=0)
   1312 
   1313     def _getitem_axis(self, key, axis=0):

/usr/local/lib/python3.5/dist-packages/pandas/core/indexing.py in _getitem_axis(self, key, axis)
   1479 
   1480         # fall thru to straight lookup
-> 1481         self._has_valid_type(key, axis)
   1482         return self._get_label(key, axis=axis)
   1483 

/usr/local/lib/python3.5/dist-packages/pandas/core/indexing.py in _has_valid_type(self, key, axis)
   1406 
   1407             try:
-> 1408                 key = self._convert_scalar_indexer(key, axis)
   1409                 if key not in ax:
   1410                     error()

/usr/local/lib/python3.5/dist-packages/pandas/core/indexing.py in _convert_scalar_indexer(self, key, axis)
    193         ax = self.obj._get_axis(min(axis, self.ndim - 1))
    194         # a scalar
--> 195         return ax._convert_scalar_indexer(key, kind=self.name)
    196 
    197     def _convert_slice_indexer(self, key, axis):

/usr/local/lib/python3.5/dist-packages/pandas/indexes/base.py in _convert_scalar_indexer(self, key, kind)
   1169             elif kind in ['loc'] and is_integer(key):
   1170                 if not self.holds_integer():
-> 1171                     return self._invalid_indexer('label', key)
   1172 
   1173         return key

/usr/local/lib/python3.5/dist-packages/pandas/indexes/base.py in _invalid_indexer(self, form, key)
   1282                         "indexers [{key}] of {kind}".format(
   1283                             form=form, klass=type(self), key=key,
-> 1284                             kind=type(key)))
   1285 
   1286     def get_duplicates(self):

TypeError: cannot do label indexing on <class 'pandas.indexes.base.Index'> with these indexers [1] of <class 'int'>
```

In [70]:

```
df.iloc[0]
```

Out[70]:

```
one              1
bar              1
flag         False
foo            bar
one_trunc        1
Name: a, dtype: object
```

In [71]:

```
df.loc['b'] #row with label b
```

Out[71]:

```
one              2
bar              2
flag         False
foo            bar
one_trunc        2
Name: b, dtype: object
```

In [72]:

```
df.iloc[3] #fourth row
```

Out[72]:

```
one            NaN
bar            NaN
flag         False
foo            bar
one_trunc      NaN
Name: d, dtype: object
```

Data alignment

- two DataFrame objects automatically align on both the columns and the index (row labels)
- the resulting object will have the union of the column and row labels

In [73]:

```
df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])
df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])
```

In [74]:

```
df
```

Out[74]:

	A	B	C	D
0	-0.484364	-0.016477	0.967994	0.283003
1	-0.390065	-0.972981	0.774240	0.810957
2	0.285882	-0.429388	-0.975174	-0.110529
3	1.727948	0.999428	0.201771	-0.058377
4	-0.144042	-0.487939	2.775026	0.467772
5	0.932363	-1.117442	0.239351	0.154553
6	-0.079045	0.758972	1.177784	0.831585
7	-1.124921	-0.787399	0.177390	-1.890339
8	0.495383	-1.437699	-1.547658	-0.694165
9	-1.261681	1.399010	-0.361245	-0.048354

In [75]:

```
df2
```

Out[75]:

	A	B	C
0	0.940779	-0.957628	-1.071097
1	0.485437	-0.028251	-1.838171
2	-0.772219	1.523176	1.407595
3	-0.000005	-0.125296	-1.027750
4	1.633078	0.206675	0.133854
5	0.185198	-0.991530	-1.030470
6	0.077898	-0.426124	-0.994866

In [76]:

```
df + df2
```

Out[76]:

	A	B	C	D
0	0.456415	-0.974105	-0.103103	NaN
1	0.095371	-1.001232	-1.063930	NaN
2	-0.486337	1.093788	0.432421	NaN
3	1.727943	0.874132	-0.825979	NaN
4	1.489036	-0.281265	2.908880	NaN
5	1.117562	-2.108972	-0.791118	NaN
6	-0.001147	0.332848	0.182917	NaN
7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN

When doing an operation between DataFrame and Series, the default behavior is to align the Series index on the DataFrame columns, thus broadcasting row-wise:

In [76]:

```
df - df.iloc[0]
```

Out[76]:

	A	B	C	D
0	0.000000	0.000000	0.000000	0.000000
1	-1.898469	-0.063016	1.337370	-0.912623
2	-0.249098	0.857885	0.888590	-1.927139
3	0.909986	2.939598	1.259622	-0.708306
4	1.321939	0.342238	0.514161	1.802644
5	-1.169097	1.634311	1.026459	2.134323
6	0.076419	1.725610	2.088969	0.576398
7	2.225730	3.866714	1.509470	1.278326
8	1.865831	1.623380	0.985613	-1.441773
9	0.547709	2.415072	-0.039689	-0.121213

In the special case of working with time series data, and the DataFrame index also contains dates, the broadcasting will be column-wise:

In [77]:

```
index = pd.date_range('1/1/2000', periods=8)
index
```

Out[77]:

```
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-0
4',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-0
8'],
              dtype='datetime64[ns]', freq='D')
```

In [78]:

```
df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=list('ABC'))
df
```

Out[78]:

	A	B	C
2000-01-01	1.063888	0.417752	-0.037131
2000-01-02	-0.423543	-0.599495	-0.132575
2000-01-03	-0.078154	-0.312301	0.714297
2000-01-04	0.218235	0.474230	1.146965
2000-01-05	0.784061	0.843337	-0.943789
2000-01-06	-0.709524	0.398018	-0.443066
2000-01-07	-0.389967	-0.156301	-1.201511
2000-01-08	0.304356	0.292492	-0.091346

In [77]:

```
df['A']
```

Out[77]:

```
0   -0.484364
1   -0.390065
2    0.285882
3    1.727948
4   -0.144042
5    0.932363
6   -0.079045
7   -1.124921
8    0.495383
9   -1.261681
Name: A, dtype: float64
```

In [79]:

```
df - df['A']
```

Out[79]:

	2000-01-01 00:00:00	**2000-01-02 00:00:00**	**2000-01-03 00:00:00**	**2000-01-04 00:00:00**	**2000-01-05 00:00:00**	**2000-01-06 00:00:00**	**2000-01-07 00:00:00**	**2000-01-08 00:00:00**
2000-01-01	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-02	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-03	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-04	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-05	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-06	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-08	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Thus, it is better to define the alignment axis:

In [80]:

```
df.sub(df['A'], axis=0)
```

Out[80]:

	A	**B**	**C**
2000-01-01	0.0	-0.646136	-1.101020
2000-01-02	0.0	-0.175952	0.290968
2000-01-03	0.0	-0.234147	0.792451
2000-01-04	0.0	0.255995	0.928729
2000-01-05	0.0	0.059276	-1.727850
2000-01-06	0.0	1.107542	0.266458
2000-01-07	0.0	0.233666	-0.811544
2000-01-08	0.0	-0.011865	-0.395702

In [81]:

```
df.sub(df['A'], axis=1)
```

Out[81]:

	2000-01-01 00:00:00	2000-01-02 00:00:00	2000-01-03 00:00:00	2000-01-04 00:00:00	2000-01-05 00:00:00	2000-01-06 00:00:00	2000-01-07 00:00:00	2000-01-08 00:00:00
2000-01-01	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-02	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-03	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-04	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-05	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-06	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-08	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Operations with scalars and boolean operations

Operations with scalars work element-wise:

In [78]:

```
df
```

Out[78]:

	A	B	C	D
0	-0.484364	-0.016477	0.967994	0.283003
1	-0.390065	-0.972981	0.774240	0.810957
2	0.285882	-0.429388	-0.975174	-0.110529
3	1.727948	0.999428	0.201771	-0.058377
4	-0.144042	-0.487939	2.775026	0.467772
5	0.932363	-1.117442	0.239351	0.154553
6	-0.079045	0.758972	1.177784	0.831585
7	-1.124921	-0.787399	0.177390	-1.890339
8	0.495383	-1.437699	-1.547658	-0.694165
9	-1.261681	1.399010	-0.361245	-0.048354

In [79]:

```
df * 5 + 1
```

Out[79]:

	A	B	C	D
0	-1.421821	0.917614	5.839971	2.415016
1	-0.950327	-3.864904	4.871202	5.054786
2	2.429410	-1.146938	-3.875870	0.447357
3	9.639739	5.997140	2.008855	0.708117
4	0.279791	-1.439697	14.875131	3.338860
5	5.661816	-4.587209	2.196757	1.772765
6	0.604773	4.794861	6.888918	5.157926
7	-4.624604	-2.936997	1.886951	-8.451695
8	3.476913	-6.188496	-6.738292	-2.470827
9	-5.308404	7.995050	-0.806226	0.758230

In [80]:

```
df ** 2
```

Out[80]:

	A	B	C	D
0	0.234609	0.000272	0.937013	0.080091
1	0.152151	0.946692	0.599448	0.657652
2	0.081729	0.184374	0.950964	0.012217
3	2.985804	0.998856	0.040712	0.003408
4	0.020748	0.238085	7.700770	0.218811
5	0.869301	1.248676	0.057289	0.023887
6	0.006248	0.576039	1.387174	0.691534
7	1.265447	0.619998	0.031467	3.573382
8	0.245404	2.066979	2.395246	0.481866
9	1.591838	1.957229	0.130498	0.002338

Boolean operators work as well:

In [81]:

```
df1 = pd.DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] }, dtype=bool)
df1
```

Out[81]:

	a	b
0	True	False
1	False	True
2	True	True

In [82]:

```
df2 = pd.DataFrame({'a' : [0, 1, 1], 'b' : [1, 1, 0] }, dtype=bool)
df2
```

Out[82]:

	a	b
0	False	True
1	True	True
2	True	False

In [83]:

```
df1 & df2 # and
```

Out[83]:

	a	b
0	False	False
1	False	True
2	True	False

In [84]:

```
df1 | df2 # or
```

Out[84]:

	a	b
0	True	True
1	True	True
2	True	True

In [85]:

```
df1 ^ df2 # xor
```

Out[85]:

	a	b
0	True	True
1	True	False
2	False	True

In [86]:

```
-df1
```

Out[86]:

	a	b
0	False	True
1	True	False
2	False	False

Transposing

Similar to ndarray, we can access the `T` attribute of a DataFrame or use the `transpose` function:

In [87]:

```
df[:5].T
```

Out[87]:

	0	1	2	3	4
A	-0.484364	-0.390065	0.285882	1.727948	-0.144042
B	-0.016477	-0.972981	-0.429388	0.999428	-0.487939
C	0.967994	0.774240	-0.975174	0.201771	2.775026
D	0.283003	0.810957	-0.110529	-0.058377	0.467772

DataFrame interoperability with NumPy

Elementwise NumPy ufuncs (log, exp, sqrt, ...) and various other NumPy functions can be used with no issues on DataFrame, assuming the data within are numeric:

In [92]:

```
np.exp(df)
```

Out[92]:

	A	B	C
2000-01-01	2.897616	1.518544	0.963550
2000-01-02	0.654723	0.549089	0.875837
2000-01-03	0.924822	0.731761	2.042750
2000-01-04	1.243880	1.606777	3.148621
2000-01-05	2.190349	2.324110	0.389151
2000-01-06	0.491878	1.488871	0.642065
2000-01-07	0.677079	0.855302	0.300739
2000-01-08	1.355752	1.339761	0.912702

In [93]:

```
np.asarray(df)
```

Out[93]:

```
array([[ 1.06388832,  0.41775211, -0.03713139],
       [-0.42354263, -0.59949456, -0.13257504],
       [-0.07815422, -0.3123009 ,  0.71429685],
       [ 0.21823541,  0.47423045,  1.14696457],
       [ 0.78406095,  0.8433372 , -0.94378863],
       [-0.70952417,  0.39801805, -0.44306629],
       [-0.3899669 , -0.156301  , -1.20151131],
       [ 0.30435621,  0.29249157, -0.09134566]])
```

In [94]:

```
df.T.dot(df)
```

Out[94]:

	A	B	C
A	2.727865	1.355053	0.226257
B	1.355053	1.835954	-0.426385
C	0.226257	-0.426385	4.383721

DataFrame info

In [88]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 4 columns):
A    10 non-null float64
B    10 non-null float64
C    10 non-null float64
D    10 non-null float64
dtypes: float64(4)
memory usage: 400.0 bytes
```

Text representation

In [89]:

```
print(df.to_string())
```

```
          A         B         C         D
0 -0.484364 -0.016477  0.967994  0.283003
1 -0.390065 -0.972981  0.774240  0.810957
2  0.285882 -0.429388 -0.975174 -0.110529
3  1.727948  0.999428  0.201771 -0.058377
4 -0.144042 -0.487939  2.775026  0.467772
5  0.932363 -1.117442  0.239351  0.154553
6 -0.079045  0.758972  1.177784  0.831585
7 -1.124921 -0.787399  0.177390 -1.890339
8  0.495383 -1.437699 -1.547658 -0.694165
9 -1.261681  1.399010 -0.361245 -0.048354
```

Basic functionality

In [144]:

```
index = pd.date_range('1/1/2000', periods=8)

s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

df = pd.DataFrame(np.random.randn(8, 3),index=index,columns=['A', 'B', 'C'])
```

Head and tail

- to view a small sample of a Series or DataFrame object
- default number of elements to display is five
- a custom number may be passed

In [145]:

```
long_series = pd.Series(np.random.randn(1000))
```

In [146]:

```
long_series.head()
```

Out[146]:

```
0   -0.951095
1    2.066487
2    1.810913
3    1.039186
4    1.989254
dtype: float64
```

In [147]:

```
long_series.tail()
```

Out[147]:

```
995    0.504562
996   -1.213206
997   -0.929903
998    2.315452
999   -0.160619
dtype: float64
```

In [148]:

```
long_series.tail(4)
```

Out[148]:

```
996   -1.213206
997   -0.929903
998    2.315452
999   -0.160619
dtype: float64
```

Attributes of the objects

In [149]:

```
s.values #data inside the data structure
```

Out[149]:

```
array([ 2.74681418,  1.39881223, -1.21507742, -0.05904648,  0.430317
31])
```

In [150]:

```
df.values
```

Out[150]:

```
array([[-0.33455454, -0.38274835,  0.52926316],
       [-2.25720573,  0.19985595,  0.26127717],
       [-1.25641287,  0.27889226,  0.47381959],
       [-0.72228451,  1.25939948, -0.80087508],
       [ 1.56510487,  1.37449467, -0.06186954],
       [-1.48897772, -0.15255919, -0.74840904],
       [-1.53220742, -1.42835346,  0.17823569],
       [ 1.14101045, -0.09312767,  1.46111598]])
```

Missing data

In [123]:

```
df
```

Out[123]:

	A	B	C
0	0.206250	-1.204279	-1.134898
1	0.985401	-0.184833	-0.612037
2	3.647275	-1.389859	0.046891
3	0.109514	0.296153	0.603235

In [117]:

```
df2  = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])
```

In [118]:

```
df2
```

Out[118]:

	A	B	C
0	-0.253998	0.482435	-1.726536
1	0.953165	1.772352	1.238428
2	2.287791	0.120891	-0.054541
3	0.341455	0.996974	-1.313491
4	2.394826	1.456366	0.108582
5	1.571921	-1.057910	0.661252
6	0.805154	-0.218137	-0.795221

In [119]:

```
df + df2
```

Out[119]:

	A	B	C
0	0.936697	-0.892587	-3.250157
1	3.585261	2.664543	0.662621
2	2.508377	0.132018	-0.210761
3	-0.381379	0.706569	-0.851599
4	NaN	NaN	NaN
5	NaN	NaN	NaN
6	NaN	NaN	NaN

- arithmetic functions have the option of inputting a fill_value (value to substitute when at most one of the values at a location are missing)

In [120]:

```
df.add(df2, fill_value=0)
```

Out[120]:

	A	B	C
0	0.936697	-0.892587	-3.250157
1	3.585261	2.664543	0.662621
2	2.508377	0.132018	-0.210761
3	-0.381379	0.706569	-0.851599
4	2.394826	1.456366	0.108582
5	1.571921	-1.057910	0.661252
6	0.805154	-0.218137	-0.795221

Comparisons

In [121]:

```
df.gt(df2)
```

Out[121]:

	A	B	C
0	True	False	True
1	True	False	False
2	False	False	False
3	False	False	True
4	False	False	False
5	False	False	False
6	False	False	False

In [122]:

```
df2.gt(df)
```

Out[122]:

	A	B	C
0	False	True	False
1	False	True	True
2	True	True	True
3	True	True	False
4	False	False	False
5	False	False	False
6	False	False	False

In [123]:

```
df2.ne(df)
```

Out[123]:

	A	B	C
0	True	True	True
1	True	True	True
2	True	True	True
3	True	True	True
4	True	True	True
5	True	True	True
6	True	True	True

Boolean reductions

In [124]:

```
(df > 0).all()
```

Out[124]:

```
A    False
B    False
C    False
dtype: bool
```

In [125]:

```
(df > 0).any()
```

Out[125]:

```
A    True
B    True
C    True
dtype: bool
```

In [126]:

```
(df > 0).any().any() #final boolean value
```

Out[126]:

```
True
```

In [134]:

```
df.empty
```

Out[134]:

```
False
```

In [135]:

```
pd.DataFrame(columns=list('ABC')).empty
```

Out[135]:

```
True
```

In [136]:

```
pd.Series([True]).bool()
```

Out[136]:

```
True
```

In [137]:

```
pd.Series([False]).bool()
```

Out[137]:

```
False
```

Warning!

In [127]:

```
if df:
    pass
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-127-c1517edf3843> in <module>()
----> 1 if df:
      2     pass

/usr/local/lib/python3.5/dist-packages/pandas/core/generic.py in __nonzero__(self)
    915         raise ValueError("The truth value of a {0} is ambiguous. "
    916                          "Use a.empty, a.bool(), a.item(), a.any() or a.all()."
--> 917                          .format(self.__class__.__name__))
    918 
    919     __bool__ = __nonzero__

ValueError: The truth value of a DataFrame is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all().
```

Comparing if objects are equivalent

In [138]:

```
data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data2)
```

In [139]:

```
df
```

Out[139]:

	a	b	c
0	1	2	NaN
1	5	10	20.0

In [140]:

```
df + df
```

Out[140]:

	a	b	c
0	2	4	NaN
1	10	20	40.0

In [141]:

```
df * 2
```

Out[141]:

	a	b	c
0	2	4	NaN
1	10	20	40.0

In [142]:

```
df + df == df * 2
```

Out[142]:

	a	b	c
0	True	True	False
1	True	True	True

In [143]:

```
np.nan == np.nan
```

Out[143]:

```
False
```

In [144]:

```
(df+df == df*2).all()
```

Out[144]:

```
a     True
b     True
c    False
dtype: bool
```

In [145]:

```
(df+df).equals(df*2)
```

Out[145]:

```
True
```

Comparing with scalar values

In [128]:

```
pd.Series(['foo', 'bar', 'baz']) == 'foo'
```

Out[128]:

```
0     True
1    False
2    False
dtype: bool
```

In [129]:

```
pd.Index(['foo', 'bar', 'baz']) == 'foo'
```

Out[129]:

```
array([ True, False, False], dtype=bool)
```

In [130]:

```
pd.Series(['foo', 'bar', 'baz']) == pd.Index(['foo', 'bar', 'qux'])
```

Out[130]:

```
0     True
1     True
2    False
dtype: bool
```

In [131]:

```
pd.Series(['foo', 'bar', 'baz']) == np.array(['foo', 'bar', 'qux'])
```

Out[131]:

```
0     True
1     True
2    False
dtype: bool
```

In [132]:

```
pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo', 'bar'])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-132-5f0e4fe0d85e> in <module>()
----> 1 pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo', 'bar'])

/usr/local/lib/python3.5/dist-packages/pandas/core/ops.py in wrapper(self, other, axis)
    810             if not self._indexed_same(other):
    811                 msg = 'Can only compare identically-labeled Series objects'
--> 812                 raise ValueError(msg)
    813             return self._constructor(na_op(self.values, other.values),
    814                                      index=self.index, name=name)

ValueError: Can only compare identically-labeled Series objects
```

Combining overlapping datasets

- two similar data sets where values in one are preferred over the other
- we would like to combine two DataFrame objects where missing values in one DataFrame are conditionally filled with like-labeled values from the other DataFrame

In [135]:

```
df1 = pd.DataFrame({'A' : [1., np.nan, 3., 5., np.nan],
                    'B' : [np.nan, 2., 3., np.nan, 6.]})
```

In [136]:

```
df2 = pd.DataFrame({'A' : [5., 2., 4., np.nan, 3., 7.],
                   'B' : [np.nan, np.nan, 3., 4., 6., 8.]})
```

In [137]:

```
df1
```

Out[137]:

	A	B
0	1.0	NaN
1	NaN	2.0
2	3.0	3.0
3	5.0	NaN
4	NaN	6.0

In [138]:

```
df2
```

Out[138]:

	A	B
0	5.0	NaN
1	2.0	NaN
2	4.0	3.0
3	NaN	4.0
4	3.0	6.0
5	7.0	8.0

In [139]:

```
df1.combine_first(df2)
```

Out[139]:

	A	B
0	1.0	NaN
1	2.0	2.0
2	3.0	3.0
3	5.0	4.0
4	3.0	6.0
5	7.0	8.0

Using a general `combine` method gives us full control over the process:

In [140]:

```
combiner = lambda x, y: np.where(pd.isnull(x), y, x)
df1.combine(df2, combiner)
```

Out[140]:

	A	B
0	1.0	NaN
1	2.0	2.0
2	3.0	3.0
3	5.0	4.0
4	3.0	6.0
5	7.0	8.0

Descriptive statistics

- usually first step in the analysis of collected data
- simple summaries about the sample and about the observations that have been made
- methods:
    - quantitative summary (usually in tabular form)
    - simple-to-understand graphs (e.g. histograms)
    - measures of central tendency and measures of variability or dispersion (mean, median, mode, variance, min/max values, kurtosis, skewness)

Available functions

- most of thefunctions are aggregations (hence producing a lower-dimensional result)
- some of them, like `cumsum()` and `cumprod()`, produce an object of the same size
- usually take an axis argument (specified by name or integer):
    - **Series**: no axis argument required
    - **DataFrame**: "index" (axis=0, default), "columns" (axis=1)

Function	Description
`count`	number of non-null observations
`sum`	sum of values
`mean`	mean of values
`mad`	mean absolute deviation, $D=\frac{\sum_i \lvert x_i-\hat{x}\rvert}{n}$
`median`	arithmetic median of values
`min`	minimum
`max`	maximum
`mode`	mode (data values that appear most often)
`abs`	absolute value
`prod`	product of values
`std`	sample standard deviation
`var`	unbiased variance
`sem`	standard error of the mean, $SE = \frac{s}{\sqrt{n}}$
`skew`	sample skewness (3rd moment)
`kurt`	sample kurtosis (4th moment)
`quantile`	sample quantile (value at %)
`cumsum`	cumulative sume
`cumprod`	cummulative product
`cummax`	cummulative maximum
`cummin`	cummulative minimum

Examples

In [151]:

```
df = pd.DataFrame(np.random.randn(4, 3),
                  columns=['A', 'B', 'C'])
df
```

Out[151]:

	A	B	C
0	1.542674	-0.025468	0.343962
1	0.537489	-1.363312	-2.008847
2	-0.202073	-1.006213	0.600617
3	-1.126926	-0.237790	1.323331

In [152]:

```
df.mean() #axis=0 by default -> mean of the column
```

Out[152]:

```
A    0.187791
B   -0.658196
C    0.064766
dtype: float64
```

In [161]:

```
df.mean(1) #mean of the row
```

Out[161]:

```
0   -0.099015
1    0.144657
2    0.286177
3   -0.703157
dtype: float64
```

In [162]:

```
df.mean(0)
```

Out[162]:

```
A    0.077138
B   -0.411435
C    0.055792
dtype: float64
```

In [163]:

```
df1 = pd.DataFrame({'A' : [1., np.nan, 3., 5., np.nan],
                    'B' : [np.nan, 2., 2., np.nan, 6.]})
```

In [164]:

```
df1
```

Out[164]:

	A	B
0	1.0	NaN
1	NaN	2.0
2	3.0	2.0
3	5.0	NaN
4	NaN	6.0

In [165]:

```
df1.mean(1)
```

Out[165]:

```
0    1.0
1    2.0
2    2.5
3    5.0
4    6.0
dtype: float64
```

In [166]:

```
df1.mean(1,skipna=False)
```

Out[166]:

```
0    NaN
1    NaN
2    2.5
3    NaN
4    NaN
dtype: float64
```

In [167]:

```
df1.sum() # sum of elements
```

Out[167]:

```
A     9.0
B    10.0
dtype: float64
```

In [168]:

```
df1.sum(0,skipna=False)
```

Out[168]:

```
A   NaN
B   NaN
dtype: float64
```

In [169]:

```
df1.std() #standard deviation
```

Out[169]:

```
A    2.000000
B    2.309401
dtype: float64
```

Standardization

Combined with the broadcasting / arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation 1), very concisely:

In [153]:

```
df
```

Out[153]:

	A	B	C
0	1.542674	-0.025468	0.343962
1	0.537489	-1.363312	-2.008847
2	-0.202073	-1.006213	0.600617
3	-1.126926	-0.237790	1.323331

In [154]:

```
ts_stand = (df - df.mean()) / df.std()
```

In [155]:

```
ts_stand
```

Out[155]:

	A	B	C
0	1.197797	1.002358	0.193449
1	0.309154	-1.117034	-1.436763
2	-0.344663	-0.551323	0.371280
3	-1.162288	0.666000	0.872033

In [156]:

```
ts_stand.mean()
```

Out[156]:

```
A    0.000000e+00
B   -1.110223e-16
C    0.000000e+00
dtype: float64
```

In [157]:

```
ts_stand.std()
```

Out[157]:

```
A    1.0
B    1.0
C    1.0
dtype: float64
```

In [158]:

```
xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)
```

In [159]:

```
xs_stand
```

Out[159]:

	A	B	C
0	1.125008	-0.787820	-0.337188
1	1.119849	-0.316093	-0.803756
2	0.000601	-1.000300	0.999699
3	-0.897403	-0.180584	1.077987

In [160]:

```
xs_stand.mean(1)
```

Out[160]:

```
0    0.000000e+00
1    3.700743e-17
2    3.700743e-17
3    7.401487e-17
dtype: float64
```

In [161]:

```
xs_stand.std(1)
```

Out[161]:

```
0    1.0
1    1.0
2    1.0
3    1.0
dtype: float64
```

Cummulative functions

Cummulative functions preserve the location of NaN values:

In [162]:

```
df1
```

Out[162]:

	A	B
0	1.0	NaN
1	NaN	2.0
2	3.0	3.0
3	5.0	NaN
4	NaN	6.0

In [163]:

```
df1.cumsum()
```

Out[163]:

	A	B
0	1.0	NaN
1	NaN	2.0
2	4.0	5.0
3	9.0	NaN
4	NaN	11.0

In [164]:

```
df1.cumprod()
```

Out[164]:

	A	B
0	1.0	NaN
1	NaN	2.0
2	3.0	6.0
3	15.0	NaN
4	NaN	36.0

Unique non-null values

- only for Series objects:

In [166]:

```
series = pd.Series(np.random.randn(500))
series[20:500] = np.nan
series[10:20] = 5
series
```

Out[166]:

```
0     -1.231221
1     -0.442798
2     -0.279305
3      0.247389
4     -0.480701
5      1.179192
6      0.907943
7     -0.607358
8     -0.465843
9     -0.491132
10     5.000000
11     5.000000
12     5.000000
13     5.000000
14     5.000000
15     5.000000
16     5.000000
17     5.000000
18     5.000000
19     5.000000
20          NaN
21          NaN
22          NaN
23          NaN
24          NaN
25          NaN
26          NaN
27          NaN
28          NaN
29          NaN
         ...
470         NaN
471         NaN
472         NaN
473         NaN
474         NaN
475         NaN
476         NaN
477         NaN
478         NaN
479         NaN
480         NaN
481         NaN
482         NaN
483         NaN
484         NaN
485         NaN
486         NaN
487         NaN
488         NaN
489         NaN
490         NaN
491         NaN
492         NaN
493         NaN
494         NaN
495         NaN
496         NaN
497         NaN
498         NaN
```

```
498         NaN

499         NaN
dtype: float64
```

In [167]:

```
series.nunique()
```

Out[167]:

```
11
```

Summarizing data

- `describe()` function which computes a variety of summary statistics
- both Series and DataFrames
- NaNs excluded

In [168]:

```
series.describe()
```

Out[168]:

```
count    20.000000
mean      2.416808
std       2.698001
min      -1.231221
25%      -0.448559
50%       3.089596
75%       5.000000
max       5.000000
dtype: float64
```

In [169]:

```
frame = pd.DataFrame(np.random.randn(1000, 5), columns=['a', 'b', 'c', 'd',
'e'])
print(frame)
```

	a	b	c	d	e
0	-0.319423	-0.394861	-1.255406	-0.149172	0.683483
1	-1.345876	0.426508	-1.041345	-1.601146	0.611494
2	-0.722750	-1.108578	0.697783	-0.265983	0.164342
3	-1.130094	0.508653	1.374792	-0.339138	-0.634866
4	0.548841	-1.617609	-0.858189	0.813601	-0.808907
5	0.926488	0.944165	1.754313	-0.165347	-0.685100
6	-1.794617	-1.535325	-0.289502	-0.224489	1.096498
7	0.477868	-0.229786	-1.254461	0.271146	-0.372939
8	1.064817	1.090365	0.421944	0.328533	0.121780
9	-1.126377	-0.530643	-0.523274	0.430011	-0.325511
10	1.839823	-1.153210	0.532645	0.579230	-0.969492
11	-1.640543	-1.060738	-0.074757	-1.092711	0.200253
12	0.093289	-0.108198	-0.170994	-1.167397	-0.656845
13	-0.302590	-0.772729	0.425957	0.418983	0.520989
14	0.081489	-0.047071	1.159678	1.554944	-0.469201
15	0.788194	0.672976	-1.260406	-1.873323	-0.015046
16	-0.580251	-1.485709	0.252197	1.022355	-0.341453
17	1.103938	0.132462	-1.399502	-0.133395	1.342352
18	1.525408	-0.898846	0.574619	-1.455967	0.641993
19	-0.006003	-0.453654	-0.215742	-0.177199	-1.587829
20	-1.207371	0.574454	1.100459	1.996648	-1.052073
21	-0.089599	0.164201	-0.186620	-0.784308	0.518659
22	-0.559043	-0.449779	1.089828	-1.033932	0.242483
23	-0.367806	0.036083	0.582336	0.148006	1.058738
24	0.536012	0.831085	1.804037	0.776036	0.364344
25	0.026974	0.419113	-0.140349	-0.144612	-0.298261
26	-0.219382	0.120642	0.216115	0.494308	0.490932
27	1.669894	-1.184396	-1.053520	-0.027844	0.161601
28	-0.407008	0.123008	0.195181	-1.893676	1.257980
29	-0.540863	1.408090	0.540220	-0.661260	0.734481
..	...	...	...	...	...
970	0.543373	0.092504	1.102079	-1.482767	-0.724965
971	0.382068	0.732826	-2.164029	0.649441	-1.013747
972	0.143462	-0.340539	0.672197	1.381635	0.150014
973	-1.145670	-0.145782	1.266940	1.743497	-1.812559
974	2.218413	0.716531	1.426966	-0.993612	1.166308
975	-1.063589	-1.592966	0.279819	0.270352	1.789851
976	-2.154921	-0.026491	0.814728	-0.044263	-1.026778
977	1.095930	0.856255	-0.504769	-0.021840	-1.233613
978	0.318225	-0.131976	-0.735600	0.416457	0.705491
979	0.677742	1.864300	-0.986692	-0.104328	0.902466
980	1.807447	-0.063977	0.691202	-0.289349	-0.943867
981	0.499583	1.386752	1.165266	0.316804	2.105193
982	0.164896	0.102640	1.768660	0.610850	0.004691
983	-0.926024	-0.121243	-1.550775	0.384594	-1.827523
984	-0.415864	1.817988	1.700908	0.252386	0.358304
985	-0.488019	-0.137039	0.150522	-1.249176	0.353840
986	-0.196000	0.497296	-0.794836	-0.816253	0.584718
987	0.905836	-0.293453	-0.868495	-0.948187	1.023070
988	-1.197001	0.592210	-0.240267	1.497983	-0.184252
989	-1.481664	-0.430472	-1.220774	-1.888897	1.094239
990	-1.343064	0.288921	-1.220118	1.420821	-0.290532
991	-0.331457	-1.003968	0.614218	0.237742	-0.439767
992	1.975992	0.835786	1.125830	0.616335	0.646331
993	1.262150	-0.000324	0.666841	-0.389231	0.296741
994	-0.572768	-1.216233	0.943629	2.024895	-0.464152
995	-0.224083	0.058527	-1.326557	-0.568194	-3.073780
996	0.104416	-0.901419	-1.509272	0.842081	-0.421302
997	0.110910	0.119223	-0.418484	1.442428	0.700481
998	0.959377	-1.709271	0.285530	-0.927774	-0.325806

```
999  1.680879 -1.401576  0.892939  1.030161 -0.070119

[1000 rows x 5 columns]
```

In [170]:

```
frame.describe()
```

Out[170]:

	a	b	c	d	e
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.011536	-0.049048	-0.067460	0.006945	-0.063300
std	0.993034	0.980171	1.038315	0.995852	0.987689
min	-3.051857	-2.859983	-3.330851	-2.890607	-3.073780
25%	-0.682852	-0.723687	-0.754955	-0.659997	-0.768263
50%	-0.039071	-0.094184	-0.055856	-0.009406	-0.130583
75%	0.679338	0.592566	0.631209	0.641419	0.597029
max	3.543436	3.305835	2.936450	3.826159	3.815609

You can select specific percentiles to include in the output:

In [189]:

```
frame.describe(percentiles=[.05, .25, .75, .95])
```

Out[189]:

	a	b	c	d	e
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.033084	0.002939	-0.012685	0.091746	-0.041505
std	1.029094	0.998507	0.998845	1.000250	0.962523
min	-2.849531	-3.078985	-3.190476	-2.787399	-3.144524
5%	-1.689548	-1.682921	-1.639631	-1.522753	-1.620122
25%	-0.733862	-0.653913	-0.665246	-0.615160	-0.695180
50%	-0.066985	0.026361	-0.009807	0.078909	-0.048173
75%	0.691474	0.690368	0.653621	0.790403	0.624621
95%	1.658014	1.648397	1.642484	1.745633	1.479933
max	3.033060	3.385996	3.116731	3.247135	3.742124

`describe()` works for non-numerical values as well:

In [171]:

```
s = pd.Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])
s
```

Out[171]:

```
0      a
1      a
2      b
3      b
4      a
5      a
6    NaN
7      c
8      d
9      a
dtype: object
```

In [172]:

```
s.describe()
```

Out[172]:

```
count     9
unique    4
top       a
freq      5
dtype: object
```

In case of mixed data only the numerical data will be summarized by default:

In [173]:

```
frame = pd.DataFrame({'a': ['Yes', 'Yes', 'No', 'No'], 'b': range(4)})
frame
```

Out[173]:

	a	b
0	Yes	0
1	Yes	1
2	No	2
3	No	3

In [174]:

```
frame.describe()
```

Out[174]:

	b
count	4.000000
mean	1.500000
std	1.290994
min	0.000000
25%	0.750000
50%	1.500000
75%	2.250000
max	3.000000

The default behavior can be however changed:

In [175]:

```
frame.describe(include='all')
```

Out[175]:

	a	b
count	4	4.000000
unique	2	NaN
top	No	NaN
freq	2	NaN
mean	NaN	1.500000
std	NaN	1.290994
min	NaN	0.000000
25%	NaN	0.750000
50%	NaN	1.500000
75%	NaN	2.250000
max	NaN	3.000000

Index of Min/Max values

In [176]:

```
s1 = pd.Series(np.random.randn(5))
s1
```

Out[176]:

```
0    0.217265
1    0.125427
2    1.095033
3   -1.600463
4   -0.458865
dtype: float64
```

In [177]:

```
s1.idxmin(), s1.idxmax()
```

Out[177]:

```
(3, 2)
```

In [178]:

```
df1 = pd.DataFrame(np.random.randn(5,3), columns=['A','B','C'])
df1
```

Out[178]:

	A	B	C
0	1.102520	0.730736	0.077419
1	-1.138234	-0.769791	-0.666982
2	0.013109	0.762810	-0.908731
3	0.403665	0.510627	0.481493
4	-0.179407	0.389031	-1.365326

In [179]:

```
df1.idxmin(axis=0)
```

Out[179]:

```
A    1
B    1
C    4
dtype: int64
```

In [180]:

```
df1.idxmax(axis=1)
```

Out[180]:

```
0    A
1    C
2    B
3    B
4    B
dtype: object
```

Value counts

In [200]:

```
data = np.random.randint(0, 7, size=50)
data
```

Out[200]:

```
array([6, 4, 3, 5, 0, 6, 5, 1, 1, 1, 6, 4, 2, 6, 2, 5, 4, 5, 3, 5,
 0, 4, 3,
       1, 5, 0, 5, 5, 3, 6, 5, 0, 3, 5, 1, 0, 4, 6, 6, 2, 1, 6, 1,
 2, 5, 3,
       5, 0, 2, 1])
```

In [201]:

```
s = pd.Series(data)
s.value_counts()
```

Out[201]:

```
5    12
6     8
1     8
3     6
0     6
4     5
2     5
dtype: int64
```

In [202]:

```
pd.value_counts(data)
```

Out[202]:

```
5    12
6     8
1     8
3     6
0     6
4     5
2     5
dtype: int64
```

Custom functions

- **pipe()** - tablewise application
- **apply()** - row or column-wise application
- **applymap()** - elementwise application

In [181]:

```
df
```

Out[181]:

	A	B	C
0	1.542674	-0.025468	0.343962
1	0.537489	-1.363312	-2.008847
2	-0.202073	-1.006213	0.600617
3	-1.126926	-0.237790	1.323331

In [182]:

```
df.apply(lambda x: x.max() - x.min())
```

Out[182]:

```
A    2.669599
B    1.337844
C    3.332177
dtype: float64
```

In [183]:

```
df.apply(lambda x: x.max() - x.min(),axis=1)
```

Out[183]:

```
0    1.568142
1    2.546336
2    1.606831
3    2.450257
dtype: float64
```

Let us assume we wanted to extract the date where the maximum value for each column occurred:

In [184]:

```
tsdf = pd.DataFrame(np.random.randn(1000, 3), columns=['A', 'B', 'C'],
                    index=pd.date_range('1/1/2000', periods=1000))
```

In [185]:

```
tsdf.head()
```

Out[185]:

	A	B	C
2000-01-01	-0.743708	-0.817322	-0.090246
2000-01-02	0.588230	-1.206334	-1.068591
2000-01-03	0.236960	1.140246	0.340530
2000-01-04	-0.679166	-0.551898	1.301277
2000-01-05	-0.071545	0.292190	0.256356

In [186]:

```
tsdf.apply(lambda x: x.idxmax())
```

Out[186]:

```
A   2002-01-23
B   2000-12-11
C   2002-05-12
dtype: datetime64[ns]
```

Similarly, we can use Series methods to manipulate columns in a DataFrame:

In [188]:

```
tsdf[10:20] = np.nan
```