STAT606
Computing for Data Science and Statistics

Lecture 13-14: `pandas`

Pandas

Open-source library of data analysis tools

- Low-level ops implemented in Cython (C+Python=Cython, often faster)
- Database-like structures, largely similar to those available in R
- Well integrated with `numpy`/`scipy`
- Optimized for most common operations
  - E.g., vectorized operations, operations on rows of a table

> **From the documentation:** pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.

Installing `pandas`

Using `conda`:

```
conda install pandas
```

Using `pip`:

```
pip install pandas
```

From binary (not recommended):

http://pypi.python.org/pypi/pandas

Warning: a few recent updates to `pandas` have been API-breaking changes, meaning they changed one or more functions (e.g., changed the number of arguments, their default values, or other behaviors). This shouldn't be a problem for us, but you may as well check that you have the most recent version installed.

Basic Data Structures

Series: represents a one-dimensional **labeled** array

- Labeled just means that there is an index into the array
- Support vectorized operations

DataFrame: table of rows, with labeled columns

- Like a spreadsheet or an R data frame
- Support `numpy` ufuncs (provided data are numeric)

pandas Series

Can create a pandas Series from any array-like structure (e.g., Python list, numpy array, dict).

By default, indices are integers, starting from 0, just like you're used to.

```python
import pandas as pd
import numpy as np
numbers = np.random.randn(5)
s = pd.Series(numbers)
s
```

```
0   -0.318743
1    0.807948
2   -0.216362
3   -0.356014
4    1.542122
dtype: float64
```

pandas tries to infer this data type automatically.

But we can specify a different set of indices if we so choose.

```python
idx = ['a','b','c','d','e']
s = pd.Series(numbers, index=idx)
s
```

```
a   -0.318743
b    0.807948
c   -0.216362
d   -0.356014
e    1.542122
dtype: float64
```

Warning: providing too few or too many indices is a ValueError.

`pandas` Series

```
d = {'dog':3.1415,'cat':42,'bird':0,'goat':1.618}
s = pd.Series(d)
s
```

```
bird      0.0000
cat      42.0000
dog       3.1415
goat      1.6180
dtype: float64
```

Can create a series from a dictionary. Keys become indices.

```
inds = ['dog','cat','bird','goat','cthulu']
s = pd.Series(d, index=inds)
s
```

```
dog        3.1415
cat       42.0000
bird       0.0000
goat       1.6180
cthulu        NaN
dtype: float64
```

Index `'cthulu'` doesn't appear in the dictionary, so `pandas` assigns it `NaN`, the standard "missing data" symbol.

`pandas` Series

Indexing works like you're used to and supports slices, but **not** negative indexing.

```
s = pd.Series([2,3,5,7,11])
s[0]
```

```
2
```

This object has type `np.int64`

```
s[1:3]
```

```
1    3
2    5
dtype: int64
```

This object is another `pandas` Series.

```
s[-1]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-22-0e2107f91cbd> in <module>()
----> 1 s[-1]
```

`pandas` Series

```
s = pd.Series([2,3,5,7,11], index=['a','a','a','a','a'])
s
```

```
a     2
a     3
a     5
a     7
a    11
dtype: int64
```

Caution: indices need not be unique in `pandas` Series. This will only cause an error if/when you try to perform an operation that requires unique indices.

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

```
a     2
a     3
a     5
a     7
a    11
dtype: int64
```

pandas Series

```
1 s
```

```
dog        3.1415
cat       42.0000
bird       0.0000
goat       1.6180
cthulu        NaN
dtype: float64
```

```
1 s[s>0]
```

```
dog      3.1415
cat     42.0000
goat     1.6180
dtype: float64
```

Series objects are like `np.ndarray` objects, so they support all the same kinds of slice operations, but note that the indices come along with the slices.

Series objects even support most `numpy` functions that act on arrays.

```
1 s**2
```

```
dog          9.869022
cat       1764.000000
bird         0.000000
goat         2.617924
cthulu            NaN
dtype: float64
```

pandas Series

```
1 s
```

```
dog         3.1415
cat        42.0000
bird        0.0000
goat        1.6180
cthulu         NaN
dtype: float64
```

Series objects are `dict`-like, in that we can access and update entries via their keys.

```
1 s['goat']
```

```
1.6180000000000001
```

```
1 s['cthulu']=-1
2 s
```

```
dog         3.1415
cat        42.0000
bird        0.0000
goat        1.6180
cthulu     -1.0000
dtype: float64
```

Not shown: Series also support the `in` operator: `x in s` checks if `x` appears as an index of Series `s`. Series also supports the dictionary `get` method.

Like a dictionary, accessing a non-existent key is a `KeyError`.

```
1 s['penguin']
```

```
---------------------------------
KeyError
&lt;ipython-input-48-a7df9b66ea8a&gt;
----> 1 s['penguin']
```

Note: I cropped out a bunch of the error message, but you get the idea.

`pandas` Series

```
1  s
```

```
dog        3.1415
cat       42.0000
bird       0.0000
goat       1.6180
cthulu    -1.0000
dtype: float64
```

Entries of a Series can be of (almost) any type, and they may be mixed (e.g., some floats, some ints, some strings, etc), but they **can not** be sequences.

```
1  s['cthulu'] = (1,1)
```

```
---------------------------------------------------------------------------
ValueError
<ipython-input-50-47579d9278ca>
----> 1 s['cthulu'] = (1,1)

/Users/keith/anaconda/lib/python2.7/site-packages/pandas
    744                 # GH 6043
    745                 elif _is_scalar_indexer(indexer):
--> 746                     values[indexer] = value
    747
    748                 # if we are an exact match (ex-broac

ValueError: setting an array element with a sequence.
```

More information on indexing: https://pandas.pydata.org/pandas-docs/stable/indexing.html

`pandas` Series

```
s
```

```
dog          3.1415
cat              42
bird              0
goat          1.618
cthulu        abcde
dtype: object
```

Series support universal functions, so long as all their entries support operations.

```
s + 2*s
```

```
dog                9.4245
cat                   126
bird                    0
goat                4.854
cthulu    abcdeabcdeabcde
dtype: object
```

```
d = {'dog':2,'cat':1.23456}
t = pd.Series(d)
t
```

```
cat    1.23456
dog    2.00000
dtype: float64
```

Series operations require that keys be shared. Missing values become `NaN` by default.

```
s+t
```

```
bird          NaN
cat       43.2346
cthulu        NaN
dog        5.1415
goat          NaN
dtype: object
```

To reiterate, Series objects support most `numpy` ufuncs. For example, `np.sqrt(s)` is valid, so long as all entries are positive.

pandas Series

```
1 s
```

```
bird     0.0000
cat     42.0000
dog      3.1415
goat     1.6180
dtype: float64
```

Series have an optional name attribute.

```
1 s.name = 'aminals'
2 s
```

```
bird     0.0000
cat     42.0000
dog      3.1415
goat     1.6180
Name: aminals, dtype: float64
```

After it is set, name attribute can be changed with rename method.

Note: this returns a new Series. It **does not** change s.name.

```
1 s.rename('animals')
```

```
bird     0.0000
cat     42.0000
dog      3.1415
goat     1.6180
Name: animals, dtype: float64
```

This will become especially useful when we start talking about DataFrames, because these name attributes will be column names.

Mapping and linking Series values

Series `map` method works analogously to Python's `map` function. Takes a function and applies it to every entry.

```
s = pd.Series(['dog','goat','skunk'])
s
```

```
0      dog
1     goat
2    skunk
dtype: object
```

```
s.map(lambda s:len(s))
```

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

Mapping and linking Series values

```
s = pd.Series(['fruit', 'animal', 'animal', 'fruit', 'fruit'],
              index=['apple','cat', 'goat','banana','kiwi'])
s
```

```
apple      fruit
cat       animal
goat      animal
banana     fruit
kiwi       fruit
dtype: object
```

```
t = pd.Series({'fruit':0,'animal':1})
s.map(t)
```

```
apple     0
cat       1
goat      1
banana    0
kiwi      0
dtype: int64
```

Series `map` also allows us to change values based on another Series. Here, we're changing the fruit/animal category labels to binary labels.

`pandas` DataFrames

Fundamental unit of `pandas`

- Analogous to R data frame

2-dimensional structure (i.e., rows and columns)

- Columns, of potentially different types
- Think: spreadsheet (or, better, database, but we haven't learned those, yet)

Can be created from many different objects

- Dict of {ndarrays, Python lists, dicts, Series}
- 2-dimensional ndarray
- Series

`pandas` DataFrames

Creating a DataFrame from a dictionary, the keys become the column names. Values become the columns of the dictionary.

```
d = {'A':pd.Series([1,2,3], index=['cat','dog','bird']),
     'B':{'cat':3.14, 'dog':2.718, 'bird':1.618, 'goat':0.5772}}
df = pd.DataFrame(d)
df
```

	A	B
bird	3.0	1.6180
cat	1.0	3.1400
dog	2.0	2.7180
goat	NaN	0.5772

Each column may have its own indices, but the resulting DataFrame will have a row for every index (i.e., every row name) that appears.

Indices that are unspecified for a given column receive `NaN`.

Note: in the code above, we specified the two columns differently. One was specified as a Series object, and the other as a dictionary. This is just to make the point that there is flexibility in how you construct your DataFrame. More options: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html

`pandas` DataFrames: creating DataFrames

Dictionary has 4 keys, so 4 columns.

```
d = {'Undergrad' : pd.Series(['UMich', 'Stanford', 'Princeton', 'Columbia'],
                              index=['Ford', 'Hoover', 'Wilson', 'Obama']),
     'PhD' : {'Wilson':'Johns Hopkins'},
     'JD' : {'Ford':'Yale','Obama':'Harvard'},
     'Terms': pd.Series([1,1,2,2], index=['Ford', 'Hoover', 'Wilson', 'Obama']) }
presidents = pd.DataFrame(d)
presidents
```

Note: Dictionary includes both text and numeric columns

	JD	PhD	Terms	Undergrad
Ford	Yale	NaN	1	UMich
Hoover	NaN	NaN	1	Stanford
Obama	Harvard	NaN	2	Columbia
Wilson	NaN	Johns Hopkins	2	Princeton

By default, rows and columns are ordered alphabetically.

`pandas` DataFrames: row/column names

	JD	PhD	Terms	Undergrad
Ford	Yale	NaN	1	UMich
Hoover	NaN	NaN	1	Stanford
Obama	Harvard	NaN	2	Columbia
Wilson	NaN	Johns Hopkins	2	Princeton

```
presidents.columns
```

```
Index([u'JD', u'PhD', u'Terms', u'Undergrad'], dtype='object')
```

```
presidents.index
```

```
Index([u'Ford', u'Hoover', u'Obama', u'Wilson'], dtype='object')
```

Row and column names accessible as the `index` and column `attributes`, respectively, of the DataFrame.

Both are returned as `pandas` Index objects.

pandas DataFrames: accessing/adding columns

```
1 presidents['PhD']
```

```
Ford                  NaN
Hoover                NaN
Obama                 NaN
Wilson     Johns Hopkins
Name: PhD, dtype: object
```

DataFrame acts like a dictionary whose keys are column names, values are Series.

```
1 type(presidents['PhD'])
```

```
pandas.core.series.Series
```

```
1 presidents['Years'] = 4*presidents['Terms']
2 presidents
```

Like a dictionary, we can create new key-value pairs.

	JD	PhD	Terms	Undergrad	Years
Ford	Yale	NaN	1	UMich	4
Hoover	NaN	NaN	1	Stanford	4
Obama	Harvard	NaN	2	Columbia	8
Wilson	NaN	Johns Hopkins	2	Princeton	8

Note: technically, this isn't quite correct, because Ford did not serve a full term. https://en.wikipedia.org/wiki/Gerald_Ford

`pandas` DataFrames: accessing/adding columns

	JD	PhD	Terms	Undergrad	Years
Ford	Yale	NaN	1	UMich	4
Hoover	NaN	NaN	1	Stanford	4
Obama	Harvard	NaN	2	Columbia	8
Wilson	NaN	Johns Hopkins	2	Princeton	8

```
presidents['Nobels'] = [0,0,1,1]
presidents
```

Since the row labels are ordered, we can specify a new column directly from a Python list, `numpy` array, etc. without having to specify indices.

	JD	PhD	Terms	Undergrad	Years	Nobels
Ford	Yale	NaN	1	UMich	4	0
Hoover	NaN	NaN	1	Stanford	4	0
Obama	Harvard	NaN	2	Columbia	8	1
Wilson	NaN	Johns Hopkins	2	Princeton	8	1

Note: by default, new column are inserted at the end. See the `insert` method to change this behavior: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.insert.html

pandas DataFrames: accessing/adding columns

	JD	PhD	Terms	Undergrad	Nobels	Years
Ford	Yale	NaN	1	UMich	0	4
Hoover	NaN	NaN	1	Stanford	0	4
Obama	Harvard	NaN	2	Columbia	1	8
Wilson	NaN	Johns Hopkins	2	Princeton	1	8

```
presidents['Fields Medals'] = 0
presidents
```

Scalars are broadcast across the rows.

	JD	PhD	Terms	Undergrad	Nobels	Years	Fields Medals
Ford	Yale	NaN	1	UMich	0	4	0
Hoover	NaN	NaN	1	Stanford	0	4	0
Obama	Harvard	NaN	2	Columbia	1	8	0
Wilson	NaN	Johns Hopkins	2	Princeton	1	8	0

Deleting columns

	JD	PhD	Terms	Undergrad	Nobels	Years	Fields Medals
Ford	Yale	NaN	1	UMich	0	4	0
Hoover	NaN	NaN	1	Stanford	0	4	0
Obama	Harvard	NaN	2	Columbia	1	8	0
Wilson	NaN	Johns Hopkins	2	Princeton	1	8	0

```
del presidents['Years']
presidents
```

	JD	PhD	Terms	Undergrad	Nobels	Fields Medals
Ford	Yale	NaN	1	UMich	0	0
Hoover	NaN	NaN	1	Stanford	0	0
Obama	Harvard	NaN	2	Columbia	1	0
Wilson	NaN	Johns Hopkins	2	Princeton	1	0

```
fields = presidents.pop('Fields Medals')
```

Delete columns identically to deleting keys from a dictionary. One can use the `del` keyword, or pop a key.

Indexing and selection

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0
Obama	Harvard	NaN	2	Columbia	1
Wilson	NaN	Johns Hopkins	2	Princeton	1

```
presidents.loc['Obama']
```

```
JD              Harvard
PhD                 NaN
Terms                 2
Undergrad      Columbia
Nobels                1
Name: Obama, dtype: object
```

```
presidents.iloc[1]
```

```
JD                  NaN
PhD                 NaN
Terms                 1
Undergrad      Stanford
Nobels                0
Name: Hoover, dtype: object
```

```
presidents['JD']
```

```
Ford          Yale
Hoover         NaN
Obama      Harvard
Wilson         NaN
Name: JD, dtype: object
```

```
presidents[1:3]
```

rms	Undergrad	Nobels
1	Stanford	0
2	Columbia	1

`df.loc` selects rows by their labels.
`df.iloc` selects rows by their integer labels (starting from 0).

```
presidents[presidents['Terms']<2]
```

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0

Indexing and selection

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0
Obama	Harvard	NaN	2	Columbia	1
Wilson	NaN	Johns Hopkins	2	Princeton	1

```
presidents
```

Select columns by their names.

```
JD                Harvard
PhD                   NaN
Terms                   2
Undergrad        Columbia
Nobels                  1
Name: Obama, dtype: object
```

```
presidents.iloc[1]
```

```
JD                    NaN
PhD                   NaN
Terms                   1
Undergrad        Stanford
Nobels                  0
Name: Hoover, dtype: object
```

```
presidents['JD']
```

```
Ford              Yale
Hoover             NaN
Obama          Harvard
Wilson             NaN
Name: JD, dtype: object
```

```
presidents[1:3]
```

	JD	PhD	Terms	Undergrad	Nobels
Hoover	NaN	NaN	1	Stanford	0
Obama	Harvard	NaN	2	Columbia	1

```
presidents[presidents['Terms']<2]
```

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0

Indexing and selection

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0
Obama	Harvard	NaN	2	Columbia	1
Wilson	NaN	Johns Hopkins	2	Princeton	1

```
1 presidents
```

```
JD
PhD
Terms
Undergrad     Columbia
Nobels
Name: Obama, d
```

```
1 presidents
```

```
JD                  NaN
PhD                 NaN
Terms                 1
Undergrad      Stanford
Nobels                0
Name: Hoover, dtype: object
```

Select rows by their numerical indices (again 0-indexed). This supports slices.

Note: one can also select slices with lists of column names, e.g., `presidents[['JD','PhD']]`.

```
1 presidents['JD']
```

```
Ford           Yale
Hoover          NaN
Obama       Harvard
Wilson          NaN
Name: JD, dtype: object
```

```
1 presidents[1:3]
```

	JD	PhD	Terms	Undergrad	Nobels
Hoover	NaN	NaN	1	Stanford	0
Obama	Harvard	NaN	2	Columbia	1

```
1 presidents[presidents['Terms']<2]
```

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0

Indexing and selection

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0
Obama	Harvard	NaN	2	Columbia	1
Wilson	NaN	Johns Hopkins	2	Princeton	1

```
presidents.loc['Obama']
```

```
JD                Harvard
PhD                   NaN
Terms                   2
Undergrad        Columbia
Nobels                  1
Name: Obama, dtype: object
```

```
presidents.iloc[1]
```

```
JD                    NaN
PhD                   NaN
Terms                   1
Undergrad        Stanford
Nobels                  0
Name: Hoover, dtype: object
```

```
presidents['JD']
```

```
Ford            Yale
Hoover           NaN
Obama        Harvard
Wilson           NaN
Name: JD, dtype: object
```

```
presidents[1:3]
```

	JD	PhD	Terms	Undergrad	Nobels
Hoover	NaN	NaN	1	Stanford	0
Obama	Harvard	NaN	2	Columbia	1

Select rows by Boolean expression.

```
presidents[presidents['Terms']<2]
```

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0

Indexing and selection

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0
Obama	Harvard	NaN	2	Columbia	1
Wilson	NaN	Johns Hopkins	2	Princeton	1

```
presidents.loc['Obama']
```

```
JD              Harvard
PhD                 NaN
Terms                 2
Undergrad      Columbia
Nobels                1
Name: Obama, dtype: object
```

```
presidents.iloc[1]
```

```
JD                  NaN
PhD                 NaN
Terms                 1
Undergrad      Stanford
Nobels                0
Name: Hoover, dtype: object
```

These expressions return Series objects.

```
presidents['JD']
```

```
Ford          Yale
Hoover         NaN
Obama      Harvard
Wilson         NaN
Name: JD, dtype: object
```

```
presidents[1:3]
```

	JD	PhD	Terms	Undergrad	Nobels
Hoover	NaN	NaN	1	Stanford	0
Obama	Harvard	NaN	2	Columbia	1

```
presidents[presidents['Terms']<2]
```

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0

Indexing and selection

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0
Obama	Harvard	NaN	2	Columbia	1
Wilson	NaN	Johns Hopkins	2	Princeton	1

```
presidents['JD']
```

```
Ford          Yale
Hoover         NaN
Obama      Harvard
Wilson         NaN
Name: JD, dtype: object
```

```
presidents.loc['Obama']
```

```
JD            Harvard
PhD               NaN
Terms               2
Undergrad    Columbia
Nobels              1
Name: Obama, dtype: object
```

These expressions return Series objects.

```
presidents.iloc[1]
```

```
JD                NaN
PhD               NaN
Terms               1
Undergrad    Stanford
Nobels              0
Name: Hoover, dtype: object
```

```
presidents[1:3]
```

	JD	PhD	Terms	Undergrad	Nobels
Hoover	NaN	NaN	1	Stanford	0
Obama	Harvard	NaN	2	Columbia	1

These expressions return DataFrames.

```
presidents[presidents['Terms']<2]
```

	JD	PhD	Terms	Undergrad	Nobels
Ford	Yale	NaN	1	UMich	0
Hoover	NaN	NaN	1	Stanford	0

More on indexing: https://pandas.pydata.org/pandas-docs/stable/indexing.html

Arithmetic with DataFrames

```
df1 = pd.DataFrame(np.random.randn(8, 4), columns=['A', 'B', 'C', 'D'])
df2 = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])
df1+df2
```

	A	B	C	D
0	0.722814	-1.889204	-1.170304	NaN
1	1.370720	-1.033425	-0.719628	NaN
2	-2.281526	0.899515	-0.298246	NaN
3	-4.276271	-2.327304	-0.444528	NaN
4	-1.418512	0.463528	0.428446	NaN
5	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN

`pandas` tries to align the DataFrames as best it can, filling in non-alignable entries with NaN.

In this example, rows 0 through 4 and columns A through C exist in both DataFrames, so these entries can be successfully added. All other entries get NaN, because `x + NaN = NaN`.

Arithmetic with DataFrames

```
df = pd.DataFrame(np.random.randn(4, 2), columns=['A', 'B'])
df
```

	A	B
0	-1.331635	-0.500870
1	1.111157	0.293138
2	-0.669850	0.456863
3	0.216643	-0.636942

By default, Series are aligned to DataFrames via row-wise broadcasting.

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

`df.iloc[0]` is a Series representing the 0-th row of `df`. When we try to subtract it from `df`, `pandas` forces dimensions to agree by broadcasting the operation across all rows of `df`.

	A	B
0	0.000000	0.000000
1	2.442791	0.794009
2	0.661785	0.957734
3	1.548277	-0.136072

Arithmetic with DataFrames

```
1 df
```

	A	B
0	-1.331635	-0.500870
1	1.111157	0.293138
2	-0.669850	0.456863
3	0.216643	-0.636942

```
1 10*df + 1
```

	A	B
0	-12.316346	-4.008702
1	12.111569	3.931385
2	-5.698497	5.568633
3	3.166428	-5.369423

Scalar addition and multiplication works in the obvious way. DataFrames also support scalar division, exponentiation… Basically every `numpy` ufunc.

DataFrames also support entrywise Boolean operations.

```
1 df > 0
```

	A	B
0	False	False
1	True	True
2	False	True
3	True	False

Arithmetic with DataFrames

	A	B
0	-1.331635	-0.500870
1	1.111157	0.293138
2	-0.669850	0.456863
3	0.216643	-0.636942

`pandas` DataFrames support `numpy`-like any and all methods.

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

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

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

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

Just like `numpy`, direct Boolean operations are not supported.

```
df or df
```

```
---------------------------------------------------------------------------
ValueError

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

Arithmetic with DataFrames

	A	B
0	-1.331635	-0.500870
1	1.111157	0.293138
2	-0.669850	0.456863
3	0.216643	-0.636942

```
df.values
```

```
array([[-1.33163456, -0.50087024],
       [ 1.11115689,  0.29313846],
       [-0.66984966,  0.45686335],
       [ 0.21664278, -0.63694229]])
```

`values` attribute stores the entries of the table in a numpy array. This is occasionally useful when you want to stop dragging the extra information around and just work with the numbers in the table.

Arithmetic with DataFrames

```
1 df
```

	A	B
0	-1.331635	-0.500870
1	1.111157	0.293138
2	-0.669850	0.456863
3	0.216643	-0.636942

DataFrames support entrywise multiplication. The `T` attribute is the transpose of the DataFrame.

```
df.T*df.T
```

	0	1	2	3
A	1.773251	1.23467	0.448699	0.046934
B	0.250871	0.08593	0.208724	0.405695

DataFrames also support matrix multiplication via the `numpy`-like `dot` method. The DataFrame dimensions must be conformal, of course.

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

	A	B
A	3.503553	0.548680
B	0.548680	0.951221

Note: Series also support a `dot` method, so you can compute inner products.

Removing NaNs

	A	B	C	D
0	-9.422331	1.100197	8.034010	NaN
1	-1.520140	5.655382	-1.692761	NaN
2	0.399654	10.058568	0.502007	NaN
3	-4.070947	2.237868	10.530079	NaN
4	1.603739	8.255591	1.892258	NaN
5	1.123450	3.141590	NaN	NaN

DataFrame `dropna` method removes rows or columns that contain NaNs.

`axis` argument controls whether we act on rows, columns, etc.

`how='any'` will remove all rows/columns that contain even one NaN. `how='all'` removes rows/columns that have all entries NaN.

```
df.dropna(axis=1,how='any')
```

	A	B
0	-9.422331	1.100197
1	-1.520140	5.655382
2	0.399654	10.058568
3	-4.070947	2.237868
4	1.603739	8.255591
5	1.123450	3.141590

```
df.dropna(axis=1,how='all')
```

	A	B	C
0	-9.422331	1.100197	8.034010
1	-1.520140	5.655382	-1.692761
2	0.399654	10.058568	0.502007
3	-4.070947	2.237868	10.530079
4	1.603739	8.255591	1.892258
5	1.123450	3.141590	NaN

Reading/writing files

`pandas` supports read/write for a wide range of different file formats. This flexibility is a major advantage of `pandas`.

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

https://pandas.pydata.org/pandas-docs/stable/io.html

Reading/writing files

`pandas` supports read/write for a wide range of different file formats. This flexibility is a major advantage of `pandas`.

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
		read_hdf	to_hdf
		read_feather	to_feather
		read_parquet	to_parquet
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

`pandas` file I/O is largely similar to R `read.table` and similar functions, so I'll leave it to you to read the `pandas` documentation as needed.

Table credit: https://pandas.pydata.org/pandas-docs/stable/io.html

Summarizing DataFrames

`pd.read_csv()` reads a comma-separated file into a DataFrame.

`info()` method prints summary data about the DataFrame. Number of rows, column names and their types, etc.

Note: there is a separate `to_string()` method that generates a string representing the DataFrame in tabular form, but this usually doesn't display well if you have many columns.

```
baseball = pd.read_csv('baseball.csv')
baseball.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21699 entries, 4 to 89534
Data columns (total 22 columns):
id       21699 non-null object
year     21699 non-null int64
stint    21699 non-null int64
team     21699 non-null object
lg       21634 non-null object
g        21699 non-null int64
ab       21699 non-null int64
r        21699 non-null int64
h        21699 non-null int64
X2b      21699 non-null int64
X3b      21699 non-null int64
hr       21699 non-null int64
rbi      21687 non-null float64
sb       21449 non-null float64
cs       17174 non-null float64
bb       21699 non-null int64
so       20394 non-null float64
ibb      14171 non-null float64
hbp      21322 non-null float64
sh       20739 non-null float64
sf       14309 non-null float64
gidp     16427 non-null float64
dtypes: float64(9), int64(10), object(3)
memory usage: 3.8+ MB
```

Summarizing DataFrames

```
1 baseball.head()
```

`head()` method displays just the first few rows of the DataFrame (5 by default; change this by supplying an argument). `tail()` displays the last few rows.

	id	year	stint	team	lg	g	ab	r	h	X2b	...	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp
4	ansonca01	1871	1	RC1	NaN	25	120	29	39	11	...	16.0	6.0	2.0	2	1.0	NaN	NaN	NaN	NaN	NaN
44	forceda01	1871	1	WS3	NaN	32	162	45	45	9	...	29.0	8.0	0.0	4	0.0	NaN	NaN	NaN	NaN	NaN
68	mathebo01	1871	1	FW1	NaN	19	89	15	24	3	...	10.0	2.0	1.0	2	0.0	NaN	NaN	NaN	NaN	NaN
99	startjo01	1871	1	NY2	NaN	33	161	35	58	5	...	34.0	4.0	2.0	3	0.0	NaN	NaN	NaN	NaN	NaN
102	suttoez01	1871	1	CL1	NaN	29	128	35	45	3	...	23.0	3.0	1.0	1	0.0	NaN	NaN	NaN	NaN	NaN

5 rows × 22 columns

Note: R and pandas both supply `head`/`tail` functions, named after UNIX/Linux commands that displays the first/last lines of a file.

Comparing DataFrames

	A	B	C	D
0	2.891255	-7.556816	-4.681215	NaN
1	5.482881	-4.133700	-2.878510	NaN
2	-9.126106	3.598060	-1.192984	NaN
3	-17.105085	-9.309218	-1.778110	NaN
4	-5.674048	1.854114	1.713784	NaN
5	NaN	NaN	NaN	NaN

These two DataFrames *ought* to be equivalent...

```
df1 = 2*df
df2 = df+df
(df1==df2).all()
```

...but they aren't.

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

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

```
False
```

```
df1.equals(df2)
```

```
True
```

Comparing DataFrames

	A	B	C	D
0	2.891255	-7.556816	-4.681215	NaN
1	5.482881	-4.133700	-2.878510	NaN
2	-9.126106	3.598060	-1.192984	NaN
3	-17.105085	-9.309218	-1.778110	NaN
4	-5.674048	1.854114	1.713784	NaN
5	NaN	NaN	NaN	NaN

These two DataFrames *ought* to be equivalent...

```
df1 = 2*df
df2 = df+df
(df1==df2).all()
```

...but they aren't.

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

The problem comes from the fact that NaNs are not equal to one another.

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

```
False
```

Solution: DataFrames have a separate `equals()` method for checking the kind of equality that we meant above.

```
df1.equals(df2)
```

```
True
```

Comparing DataFrames

	A	B	C	D
0	2.891255	-7.556816	-4.681215	NaN
1	5.482881	-4.133700	-2.878510	NaN
2	-9.126106	3.598060	-1.192984	NaN
3	-17.105085	-9.309218	-1.778110	NaN
4	-5.674048	1.854114	1.713784	NaN
5	NaN	NaN	NaN	NaN

```
df1 = 2*df
df2 = df+df
(df1==df2).all()
```

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

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

```
False
```

There is a solid design principle behind this. If there are NaNs in our data, we want to err on the side of being overly careful about what operations we perform on them. We see similar ideas in `numpy` and in R.

Solution: DataFrames have a separate `equals()` method for checking the kind of equality that we meant above.

```
df1.equals(df2)
```

```
True
```

Statistical Operations on DataFrames

	A	B	C	D
0	2.891255	-7.556816	-4.681215	NaN
1	5.482881	-4.133700	-2.878510	NaN
2	-9.126106	3.598060	-1.192984	NaN
3	-17.105085	-9.309218	-1.778110	NaN
4	-5.674048	1.854114	1.713784	NaN
5	NaN	NaN	NaN	NaN

Getting means of DataFrame rows/columns using numpy is possible, but tedious.

```
np.nanmean(np.array(df.iloc[1]))
```

```
-0.50977640954057268
```

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

```
A   -4.706220
B   -3.109512
C   -1.763407
D         NaN
dtype: float64
```

```
df.mean(1)
```

```
0   -3.115592
1   -0.509776
2   -2.240343
3   -9.397471
4   -0.702050
5         NaN
dtype: float64
```

`DataFrame.mean` method is a cleaner way to do the same thing. Argument picks out which axis to take means on: rows (1) or columns (0).

Statistical Operations on DataFrames

	A	B	C	D
0	2.			
1	5.			
2	-9.			
3	-17.			
4	-5.			
5				

ame
py is

Of course, DataFrames also support a bunch of related functions, that work similarly: `sum`, `min`, `max`, `std`, `var` etc. All of these functions take an optional Boolean argument `skipna`. If `True`, NaNs are **not included** in the computation. If `False`, NaNs are included (which can mean either that the computation doesn't work at all, or changes the value only slightly). More information: https://pandas.pydata.org/pandas-docs/stable/basics.html#descriptive-statistics

```
1 np.nanmean(np.array(df.iloc[1]))
```

```
-0.50977640954057268
```

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

```
A   -4.706220
B   -3.109512
C   -1.763407
D         NaN
dtype: float64
```

```
1 df.mean(1)
```

```
0   -3.115592
1   -0.509776
2   -2.240343
3   -9.397471
4   -0.702050
5         NaN
dtype: float64
```

`DataFrame.mean` method is a cleaner way to do the same thing. Argument picks out which axis to take means on: rows (1) or columns (0).

Summarizing DataFrames

`DataFrame.describe()` is similar to the R `summary()` function. Non-numeric data will get statistics like counts, number of unique items, etc. If a DataFrame has mixed types (both numeric and non-numeric), the non-numeric data is excluded by default.

	A	B	C	D
0	2.891255	-7.556816	-4.681215	NaN
1	5.482881	-4.133700	-2.878510	NaN
2	-9.126106	3.598060	-1.192984	NaN
3	-17.105085	-9.309218	-1.778110	NaN
4	-5.674048	1.854114	1.713784	NaN
5	NaN	NaN	NaN	NaN

```
df.describe()
```

	A	B	C	D
count	5.000000	5.000000	5.000000	0.0
mean	-4.706220	-3.109512	-1.763407	NaN
std	9.161650	5.676551	2.354438	NaN
min	-17.105085	-9.309218	-4.681215	NaN
25%	-9.126106	-7.556816	-2.878510	NaN
50%	-5.674048	-4.133700	-1.778110	NaN
75%	2.891255	1.854114	-1.192984	NaN
max	5.482881	3.598060	1.713784	NaN

Details and optional arguments:
https://pandas.pydata.org/pandas-docs/stable/basics.html#summarizing-data-describe

Row- and column-wise functions: `apply()`

	A	B	C	D
0	1.284355	1.073402	0.297575	NaN
1	-0.791592	0.841969	0.509262	NaN
2	-0.657900	-2.184139	1.635736	NaN
3	-1.897574	0.502787	-1.911790	NaN
4	0.592821	2.091333	-2.813032	NaN
5	NaN	NaN	NaN	NaN

`DataFrame.apply()` takes a function and applies it to each column of the DataFrame.

```
df.apply(np.mean)
```

```
A   -0.293978
B    0.465070
C   -0.456450
D         NaN
dtype: float64
```

`Axis` argument is 0 by default (column-wise). Change to 1 for row-wise application.

```
df.apply(np.mean, axis=1)
```

```
0    0.885111
1    0.186546
2   -0.402101
3   -1.102192
4   -0.042959
5         NaN
dtype: float64
```

Row- and column-wise functions: `apply()`

	A	B	C	D
0	1.284355	1.073402	0.297575	NaN
1	-0.791592	0.841969	0.509262	NaN
2	-0.657900	-2.184139	1.635736	NaN
3	-1.897574	0.502787	-1.911790	NaN
4	0.592821	2.091333	-2.813032	NaN
5	NaN	NaN	NaN	NaN

`Numpy` ufuncs take vectors and spit out vectors, so using `df.apply()` to apply a ufunc to every row or column in effect ends up applying the ufunc to every element.

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

	A	B	C	D
0	3.612337	2.925314	1.346589	NaN
1	0.453123	2.320931	1.664062	NaN
2	0.517938	0.112575	5.133236	NaN
3	0.149932	1.653323	0.147816	NaN
4	1.809085	8.095701	0.060023	NaN
5	NaN	NaN	NaN	NaN

Row- and column-wise functions: `apply()`

	A	B	C
0	0.938898	2.047553	-0.525091
1	1.066293	-0.599466	-0.195606
2	-0.939341	0.022376	1.453082
3	1.114664	-0.408026	-0.811081
4	2.257680	0.280994	0.847329

```
def quadratic(x, a, b, c=1):
    return a*x**2 + b*x + c
df.apply(quadratic, args=(1,2), c=5)
```

We can pass positional and keyword arguments into the function via `df.apply`. Args is a tuple of the positional arguments (in order), followed by the keyword arguments.

	A	B	C
0	7.759325	13.287581	4.225538
1	8.269566	4.160428	4.647050
2	4.003679	5.045253	10.017612
3	8.471805	4.350433	4.035691
4	14.612481	5.640946	7.412624

Note: "apply() takes an argument `raw` which is `False` by default, which converts each row or column into a Series before applying the function. When set to `True`, the passed function will instead receive an ndarray object, which has positive performance implications if you do not need the indexing functionality." This can be useful if your function is meant to work specifically with Series.

Element-wise function application

```
df = pd.DataFrame({'A': ['cat', 'dog', 'bird'],
                   'B': ['unicorn','chupacabra','pixie']})
df
```

	A	B
0	cat	unicorn
1	dog	chupacabra
2	bird	pixie

```
df.apply(lambda s:s.upper())
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent
<ipython-input-507-61f17bcd25de> in <module>()
----> 1 df.apply(lambda s:s.upper())
```

This causes an error, because `apply` thinks that its argument should be applied to Series (i.e., columns), not to individual entries.

Element-wise function application

	A	B
0	cat	unicorn
1	dog	chupacabra
2	bird	pixie

`applymap` works similarly to Python's `map` function (and the Series `map` method). Applies its argument function to every entry of the DataFrame.

```
df.applymap(lambda s:s.upper())
```

	A	B
0	CAT	UNICORN
1	DOG	CHUPACABRA
2	BIRD	PIXIE

Tablewise Function Application

Here we have a function composition applied to a DataFrame. This is perfectly valid code, but `pandas` supports another approach.

```
f = lambda x:x**2
g = lambda x:x+1
h = lambda x:2*x
df = pd.DataFrame(np.random.randn(5, 3),
                  columns=['A', 'B', 'C'])
df
```

	A	B	C
0	-2.072339	-1.282539	-1.241128
1	-0.587874	0.517591	-0.394561
2	-0.164436	1.450398	-0.975424
3	-1.215576	-0.671235	0.394053
4	-0.350299	1.958805	0.467778

```
h(g(f(df)))
```

	A	B	C
0	10.589182	5.289812	5.080798
1	2.691193	2.535802	2.311357
2	2.054078	6.207308	3.902906
3	4.955251	2.901113	2.310556
4	2.245419	9.673833	2.437633

Tablewise Function Application

The DataFrame `pipe` method is built for a pattern called **method chaining**. The `pipe` method has better support for passing additional arguments around than does the function composition to the right. This pattern using `pipe` is also more conducive to functional programming patterns.

```
f = lambda x:x**2
g = lambda x:x+1
h = lambda x:2*x
df = pd.DataFrame(np.random.randn(5, 3),
                  columns=['A', 'B', 'C'])
df
```

	A	B	C
0	-2.072339	-1.282539	-1.241128
1	-0.587874	0.517591	-0.394561
2	-0.164436	1.450398	-0.975424
3	-1.215576	-0.671235	0.394053
4	-0.350299	1.958805	0.467778

```
df.pipe(f).pipe(g).pipe(h)
```

	A	B	C
0	10.589182	5.289812	5.080798
1	2.691193	2.535802	2.311357
2	2.054078	6.207308	3.902906
3	4.955251	2.901113	2.310556
4	2.245419	9.673833	2.437633

```
h(g(f(df)))
```

	A	B	C
0	10.589182	5.289812	5.080798
1	2.691193	2.535802	2.311357
2	2.054078	6.207308	3.902906
3	4.955251	2.901113	2.310556
4	2.245419	9.673833	2.437633

Recap

pandas

$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$

Previously: basics of `pandas`

- Series and DataFrames
- Indexing, changing entries
- Function application

Next: more complicated operations

- Statistical computations
- Group-By operations
- Reshaping, stacking and pivoting

Recap

Previously: basics of `pandas`

- Series and DataFrames
- Indexing, changing entries
- Function application

Next: more complicated operations

- Statistical computations
- Group-By operations
- Reshaping, stacking and pivoting

Caveat: `pandas` is a large, complicated package, so I will not endeavor to mention every feature here. These slides should be enough to get you started, but there's no substitute for reading the documentation.

Percent change over time

`pct_change` method is supported by both Series and DataFrames. `Series.pct_change` returns a new Series representing the step-wise percent change.

Note: `pandas` has extensive support for time series data, which we mostly won't talk about in this course. Refer to the documentation for more.

```
s = pd.Series(np.random.randn(8))
s
```

```
0   -0.669520
1   -0.864352
2   -1.686718
3    0.014609
4   -2.199920
5   -0.505137
6   -0.403893
7   -0.358685
dtype: float64
```

```
s.pct_change()
```

```
0           NaN
1      0.291003
2      0.951425
3     -1.008661
4   -151.589298
5     -0.770384
6     -0.200428
7     -0.111931
dtype: float64
```

Percent change over time

`pct_change` operates on columns of a DataFrame, by default. Periods argument specifies the time-lag to use in computing percent change. So periods=2 looks at percent change compared to two time steps ago.

`pct_change` includes control over how missing data is imputed, how large a time-lag to use, etc. See documentation for more detail: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.pct_change.html

	0	1	2	3
0	-0.305249	-0.364416	0.815636	0.189141
1	2.425535	-1.082098	-0.771105	0.363440
2	-0.085443	-0.923977	-0.699232	0.897274
3	-0.116032	-0.283703	-1.372355	-1.264006
4	-0.562175	1.200134	1.039529	0.492148
5	-0.070678	-0.661320	-0.416581	0.022234

```
df.pct_change(periods=2)
```

	0	1	2	3
0	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN
2	-0.720087	1.535504	-1.857284	3.743931
3	-1.047838	-0.737821	0.779726	-4.477898
4	5.579538	-2.298878	-2.486674	-0.451508
5	-0.390876	1.331029	-0.696448	-1.017590

Computing covariances

`cov` method computes covariance between a Series and another Series.

```
s1 = pd.Series(np.random.randn(1000))
s2 = pd.Series(0.1*s1+np.random.randn(1000))
s1.cov(s2)
```

0.1522727637202401

	0	1	2	3
0	-0.305249	-0.364416	0.815636	0.189141
1	2.425535	-1.082098	-0.771105	0.363440
2	-0.085443	-0.923977	-0.699232	0.897274
3	-0.116032	-0.283703	-1.372355	-1.264006
4	-0.562175	1.200134	1.039529	0.492148
5	-0.070678	-0.661320	-0.416581	0.022234

`cov` method is also supported by DataFrame, but instead computes a new DataFrame of covariances between columns.

```
df.cov()
```

	0	1	2	3
0	1.208517	-0.515225	-0.430870	0.093096
1	-0.515225	0.673964	0.520126	-0.021969
2	-0.430870	0.520126	0.911544	0.329498
3	0.093096	-0.021969	0.329498	0.546332

`cov` supports extra arguments for further specifying behavior:
https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.cov.html

Pairwise correlations

```
df = pd.DataFrame(np.random.randn(1000, 5),
                  columns=['a', 'b', 'c', 'd', 'e'])
df.corr(method='spearman')
```

	a	b	c	d	e
a	1.000000	0.018325	-0.029441	0.002467	-0.048051
b	0.018325	1.000000	-0.000091	0.004212	-0.018435
c	-0.029441	-0.000091	1.000000	0.016103	0.034150
d	0.002467	0.004212	0.016103	1.000000	0.053519
e	-0.048051	-0.018435	0.034150	0.053519	1.000000

DataFrame `corr` method computes correlations between columns (use `axis` keyword to change this behavior). `method` argument controls which correlation score to use (default is Pearson's correlation.

```
df.corr(method='kendall')
```

	a	b	c	d	e
a	1.000000	0.012264	-0.019075	0.001333	-0.032745
b	0.012264	1.000000	0.000212	0.002515	-0.012168
c	-0.019075	0.000212	1.000000	0.009630	0.022326
d	0.001333	0.002515	0.009630	1.000000	0.035872
e	-0.032745	-0.012168	0.022326	0.035872	1.000000

Ranking data

`rank` method returns a new Series whose values are the data ranks.

Ties are broken by assigning the mean rank to both values.

```
s = pd.Series(np.random.np.random.randn(5),
              index=list('abcde'))
s
```

```
a    1.804688
b   -1.203916
c    1.055365
d   -0.048237
e    1.659330
dtype: float64
```

```
s.rank()
```

```
a    5.0
b    1.0
c    3.0
d    2.0
e    4.0
dtype: float64
```

```
s[0] = s[1] = 0
s.rank()
```

```
a    2.5
b    2.5
c    4.0
d    1.0
e    5.0
dtype: float64
```

Ranking data

	0	1	2	3	4
0	-0.606576	-0.892385	0.891247	-0.280582	0.601239
1	-1.036933	0.905388	0.012123	-2.497602	0.501482
2	0.387677	0.850437	-1.578854	-0.263305	0.540390
3	-0.631557	-0.528819	0.561295	0.955113	0.980433

By default, `rank` ranks columns of a DataFrame individually.

```
df.rank()
```

	0	1	2	3	4
0	3.0	1.0	4.0	2.0	3.0
1	1.0	4.0	2.0	1.0	1.0
2	4.0	3.0	1.0	3.0	2.0
3	2.0	2.0	3.0	4.0	4.0

Rank rows instead by supplying an `axis` argument.

```
df.rank(1)
```

	0	1	2	3	4
0	2.0	1.0	5.0	3.0	4.0
1	2.0	5.0	3.0	1.0	4.0
2	3.0	5.0	1.0	2.0	4.0
3	1.0	2.0	3.0	4.0	5.0

Note: more complicated ranking of whole rows (i.e., sorting whole rows rather than sorting columns individually) is possible, but requires we define an ordering on Series.

Aggregating data

```
tsdf = pd.DataFrame(np.random.randn(10, 3),
                    columns=['DOW', 'NASDAQ', 'S&P500'],
                    index=pd.date_range('1/1/2000', periods=10))
tsdf.head()
```

This command creates time series data, with rows indexed by year-month-day timestamps.

	DOW	NASDAQ	S&P500
2000-01-01	1.118903	0.317094	-0.936392
2000-01-02	1.091083	0.828543	-1.961891
2000-01-03	-1.309894	-1.052207	0.256100
2000-01-04	0.654260	-0.527830	0.030650
2000-01-05	-1.041396	-0.559097	0.876613

Supplying a list of functions to `agg` will apply each function to each column of the DataFrame, with each function getting a row in the resulting DataFrame.

```
tsdf.agg([np.median, np.mean, np.std])
```

	DOW	NASDAQ	S&P500
median	0.534165	0.230327	-0.076018
mean	0.391512	0.159331	-0.239343
std	1.163320	0.907218	0.773417

`agg` is an alias for the method `aggregate`. Both work exactly the same.

Aggregating data

	DOW	NASDAQ	S&P500
2000-01-01	1.118903	0.317094	-0.936392
2000-01-02	1.091083	0.828543	-1.961891
2000-01-03	-1.309894	-1.052207	0.256100
2000-01-04	0.654260	-0.527830	0.030650
2000-01-05	-1.041396	-0.559097	0.876613

`agg` can, alternatively, take a dictionary whose keys are column names, and values are functions.

```
tsdf.agg({'DOW':'mean',
          'NASDAQ':'median',
          'S&P500':'max'})
```

```
NASDAQ     0.230327
S&P500     0.876613
DOW        0.391512
dtype: float64
```

Note that the values here are strings, not functions! `pandas` supports dispatch on strings. It recognizes certain strings as referring to functions. `apply` supports similar behavior.

Aggregating data

```
df = pd.DataFrame({'A': [1, 2, 3],
                   'B': [1., 2., 3.],
                   'C': ['foo', 'bar', 'baz']})
df
```

`df` contains mixed data types.

	A	B	C
0	1	1.0	foo
1	2	2.0	bar
2	3	3.0	baz

```
df.agg(['mean', 'max'])
```

`agg` (and similarly apply) will only try to apply these functions on the columns of types supported by those functions.

	A	B	C
max	3.0	3.0	foo
mean	2.0	2.0	NaN

`pandas` doesn't know how to compute a mean string, so it doesn't try.

Note: the DataFrame `transform` method provides generally similar functionality to the `agg` method.

Iterating over Series and DataFrames

```
s
```

```
apple        fruit
cat         animal
goat        animal
banana       fruit
kiwi         fruit
dtype: object
```

```
for x in s:
    print x
```

```
fruit
animal
animal
fruit
fruit
```

Iterating over a Series gets an iterator over the values of the Series.

Iterating over a DataFrame gets an iterator over the column names.

```
df
```

	A	B	C
0	-2.072339	-1.282539	-1.241128
1	-0.587874	0.517591	-0.394561
2	-0.164436	1.450398	-0.975424
3	-1.215576	-0.671235	0.394053
4	-0.350299	1.958805	0.467778

```
for x in df:
    print x
```

```
A
B
C
```

Iterating over Series and DataFrames

```
for x in df.iteritems():
    print(x)
```

```
('A', 0   -2.072339
1   -0.587874
2   -0.164436
3   -1.215576
4   -0.350299
Name: A, dtype: float64)
('B', 0   -1.282539
1    0.517591
2    1.450398
3   -0.671235
4    1.958805
Name: B, dtype: float64)
('C', 0   -1.241128
1   -0.394561
2   -0.975424
3    0.394053
4    0.467778
Name: C, dtype: float64)
```

`iteritem()` method is supported by both Series and DataFrames. Returns an iterator over the key-value pairs. In the case of Series, these are (index,value) pairs. In the case of DataFrames, these are (colname, Series) pairs.

```
df
```

	A	B	C
0	-2.072339	-1.282539	-1.241128
1	-0.587874	0.517591	-0.394561
2	-0.164436	1.450398	-0.975424
3	-1.215576	-0.671235	0.394053
4	-0.350299	1.958805	0.467778

Iterating over Series and DataFrames

```
for x in df.iterrows():
    print(x)
```

```
(0, A   -2.072339
B   -1.282539
C   -1.241128
Name: 0, dtype: float64)
(1, A   -0.587874
B    0.517591
C   -0.394561
Name: 1, dtype: float64)
(2, A   -0.164436
B    1.450398
C   -0.975424
Name: 2, dtype: float64)
(3, A   -1.215576
B   -0.671235
C    0.394053
Name: 3, dtype: float64)
(4, A   -0.350299
B    1.958805
C    0.467778
Name: 4, dtype: float64)
```

DataFrame `iterrows()` returns an iterator over the rows of the DataFrame as (index, Series) pairs.

```
df
```

	A	B	C
0	-2.072339	-1.282539	-1.241128
1	-0.587874	0.517591	-0.394561
2	-0.164436	1.450398	-0.975424
3	-1.215576	-0.671235	0.394053
4	-0.350299	1.958805	0.467778

Iterating over Series and DataFrames

```
for x in df.iterrows():
    print(x)
```

```
(0, A   -2.072339
B   -1.282539
C   -1.241128
Name: 0, dtype: float64)
(1, A   -0.587874
B    0.517591
C   -0.394561
Name: 1, dtype: float64)
(2, A   -0.164436
B    1.450398
C   -0.975424
Name: 2, dtype: float64)
(3, A   -1.215576
B   -0.671235
C    0.394053
Name: 3, dtype: float64)
(4, A   -0.350299
B    1.958805
C    0.467778
Name: 4, dtype: float64)
```

DataFrame `iterrows()` returns an iterator over the rows of the DataFrame as (index, Series) pairs.

```
df
```

Note: DataFrames are designed to make certain operations (mainly vectorized operations) fast. This implementation has the disadvantage that iteration over a DataFrames is slow. It is usually best to avoid iterating over the elements of a DataFrame or Series, and instead find a way to compute your quantity of interest using a vectorized operation or a map/reduce operation.

4	-0.350299	1.958805	0.467778

Group By: reorganizing data

"Group By" operations are a concept from databases

- Splitting data based on some criteria
- Applying functions to different splits
- Combining results into a single data structure

Fundamental object: `pandas` GroupBy objects

Group By: reorganizing data

```python
df = pd.DataFrame({'A' : ['plant', 'animal', 'plant', 'plant'],
                   'B' : ['apple', 'goat', 'kiwi', 'grape'],
                   'C' : np.random.randn(4),
                   'D' : np.random.randn(4)})
df
```

	A	B	C	D
0	plant	apple	0.529326	-0.796997
1	animal	goat	-0.901377	-0.670747
2	plant	kiwi	1.203032	1.162924
3	plant	grape	-0.740208	1.184488

```python
df.groupby('A')
```

```
<pandas.core.groupby.DataFrameGroupBy object at 0x11fe88bd0>
```

DataFrame `groupby` method returns a `pandas groupby` object.

Group By: reorganizing data

	A	B	C	D
0	plant	apple	0.529326	-0.796997
1	animal	goat	-0.901377	-0.670747
2	plant	kiwi	1.203032	1.162924
3	plant	grape	-0.740208	1.184488

Every `groupby` object has an attribute `groups`, which is a dictionary with maps group labels to the indices in the DataFrame.

```
df.groupby('A')
```

```
<pandas.core.groupby.DataFrameGroupBy object at 0x11fe88bd0>
```

```
df.groupby('A').groups
```

```
{'animal': Int64Index([1], dtype='int64'),
 'plant': Int64Index([0, 2, 3], dtype='int64')}
```

In this example, we are splitting on the column `'A'`, which has two values: `'plant'` and `'animal'`, so the groups dictionary has two keys.

Group By: reorganizing data

	A	B	C	D
0	plant	apple	0.529326	-0.796997
1	animal	goat	-0.901377	-0.670747
2	plant	kiwi	1.203032	1.162924
3	plant	grape	-0.740208	1.184488

```
df.groupby('A')
```

```
<pandas.core.groupby.DataFrameGroupBy
```

```
df.groupby('A').groups
```

```
{'animal': Int64Index([1], dtype='int64'),
 'plant': Int64Index([0, 2, 3], dtype='int64')}
```

Every `groupby` object has an attribute `groups`, which is a dictionary with maps group labels to the indices in the DataFrame.

The important point is that the `groupby` object is storing information about how to partition the rows of the original DataFrame according to the argument(s) passed to the `groupby` method.

In this example, we are splitting on the column `'A'`, which has two values: `'plant'` and `'animal'`, so the groups dictionary has two keys.

Group By: aggregation

	A	B	C	D
0	plant	apple	0.529326	-0.796997
1	animal	goat	-0.901377	-0.670747
2	plant	kiwi	1.203032	1.162924
3	plant	grape	-0.740208	1.184488

```
1 df.groupby('A').mean()
```

A	C	D
animal	-0.901377	-0.670747
plant	0.330717	0.516805

Split on group 'A', then compute the means within each group. Note that columns for which means are not supported are removed, so column 'B' doesn't show up in the result.

Group By: aggregation

```
arrs = [['math', 'math', 'econ', 'econ', 'stats', 'stats'],
            ['left', 'right', 'left', 'right', 'left', 'right']]
index = pd.MultiIndex.from_arrays(arrs, names=['major', 'handedness'])
s = pd.Series(np.random.randn(6), index=index)
s
```

```
major  handedness
math   left         -2.015677
       right         0.537438
econ   left          1.071951
       right        -0.504158
stats  left          1.204159
       right        -0.288676
dtype: float64
```

Here we're building a hierarchically-indexed Series (i.e., multi-indexed), recording (fictional) scores of students by major and handedness.

Suppose I want to collapse over handedness to get average scores by major. In essence, I want to group by major and ignore handedness.

Group By: aggregation

```
major  handedness
math   left          -2.015677
       right          0.537438
econ   left           1.071951
       right         -0.504158
stats  left           1.204159
       right         -0.288676
dtype: float64
```

```
s.groupby(level=0).mean()
```

```
major
econ     0.283897
math    -0.739120
stats    0.457741
dtype: float64
```

Suppose I want to collapse over handedness to get average scores by major. In essence, I want to group by major and ignore handedness.

Group by the 0-th level of the hierarchy (i.e., `'major'`), and take means.

We could have equivalently written `groupby('major')`, here.

Group By: examining groups

```
1  s
```

```
major  handedness
math   left         -2.015677
       right         0.537438
econ   left          1.071951
       right        -0.504158
stats  left          1.204159
       right        -0.288676
dtype: float64
```

```
1  s.groupby('major').get_group('econ')
```

```
major  handedness
econ   left          1.071951
       right        -0.504158
dtype: float64
```

`groupby.get_group` lets us pick out an individual group. Here, we're grabbing just the data from the `'econ'` group, after grouping by `'major'`.

Group By: aggregation

Similar aggregation to what we did a few slides ago, but now we have a DataFrame instead of a Series.

major	handedness	A	B
math	left	1	-0.856890
	right	1	0.425160
econ	left	1	-0.707796
	right	1	-1.944487
stats	left	2	0.341265
	right	2	-0.938632
phys	left	3	-0.960931
	right	3	1.423622

```
1 df.groupby('handedness').mean()
```

handedness	A	B
left	1.75	-0.546088
right	1.75	-0.258584

Group By: aggregation

Similar aggregation to what we did a few slides ago, but now we have a DataFrame instead of a Series.

Groupby objects also support the `aggregate` method, which is often more convenient.

```
g = df.groupby('handedness')
g.aggregate(np.sum)
```

handedness	A	B
left	7	-2.184352
right	7	-1.034337

major	handedness	A	B
math	left	1	-0.856890
	right	1	0.425160
econ	left	1	-0.707796
	right	1	-1.944487
stats	left	2	0.341265
	right	2	-0.938632
phys	left	3	-0.960931
	right	3	1.423622

```
df.groupby('handedness').mean()
```

handedness	A	B
left	1.75	-0.546088
right	1.75	-0.258584

Transforming data

From the documentation: “The transform method returns an object that is indexed the same (same size) as the one being grouped.”

```
index = pd.date_range('10/1/1999', periods=1100)
ts = pd.Series(np.random.normal(0.5, 2, 1100), index)
ts.head()
```

```
1999-10-01   -1.283451
1999-10-02    0.468645
1999-10-03    2.796156
1999-10-04    0.449197
1999-10-05    1.647331
Freq: D, dtype: float64
```

Building a time series, indexed by year-month-day.

Suppose we want to standardize these scores within each year.

```
key = lambda d: d.year
zscore = lambda x: (x - x.mean()) / x.std()
transformed = ts.groupby(key).transform(zscore)
transformed.head()
```

```
1999-10-01   -1.097395
1999-10-02   -0.243334
1999-10-03    0.891214
1999-10-04   -0.252814
1999-10-05    0.331218
Freq: D, dtype: float64
```

Group the data according to the output of the key function, apply the given transformation within each group, then un-group the data.

Important point: the result of `groupby.transform` has the same dimension as the original DataFrame or Series.

Filtering data

```
sf = pd.Series([1, 1, 2, 2, 3, 3])
sf
```

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

```
sf.groupby(sf).filter(lambda x: x.sum() > 2)
```

```
2    2
3    2
4    3
5    3
dtype: int64
```

From the documentation: “The argument of filter must be a function that, applied to the group as a whole, returns True or False.”

So this will throw out all the groups with sum <= 2.

Like `transform`, the result is ungrouped.

Combining DataFrames

```
df1 = pd.DataFrame({'A':np.random.randn(4),
                    'B':np.random.randn(4),
                    'C':np.random.randn(4)},
                   index=[0,1,2,3])
df2 = pd.DataFrame({'A':np.random.randn(4),
                    'B':np.random.randn(4)},
                   index=[3,4,5,6])
pd.concat([df1,df2])
```

`pandas concat` function concatenates DataFrames into a single DataFrame.

	A	B	C
0	0.755669	1.497149	0.889586
1	-0.197404	0.674905	1.131785
2	0.341409	0.632993	0.495411
3	0.646052	-0.809168	-0.708263
3	0.508306	-0.070561	NaN
4	1.172885	-0.518003	NaN
5	-0.103887	-0.479715	NaN
6	0.596387	-2.156612	NaN

Repeated indices remain repeated in the resulting DataFrame.

Missing values get NaN.

`pandas.concat` accepts numerous optional arguments for finer control over how concatenation is performed. See the documentation for more.

Merges and joins

`pandas` DataFrames support many common database operations

Most notably, join and merge operations

We'll learn about these when we discuss SQL later in the semester

So we won't discuss them here

Important: What we learn for SQL later has analogues in `pandas`

If you are already familiar with SQL, you might like to read this:
https://pandas.pydata.org/pandas-docs/stable/comparison_with_sql.html

Pivoting and Stacking

	date	variable	value
0	2000-01-03	A	1.234594
1	2000-01-04	A	0.661894
2	2000-01-05	A	0.810323
3	2000-01-03	B	-0.156366
4	2000-01-04	B	0.798020
5	2000-01-05	B	-0.360506
6	2000-01-03	C	0.375464
7	2000-01-04	C	0.413346
8	2000-01-05	C	-0.071480
9	2000-01-03	D	0.108641
10	2000-01-04	D	-0.738962
11	2000-01-05	D	0.460154

Data in this format is usually called **stacked**. It is common to store data in this form in a file, but once it's read into a table, it often makes more sense to create columns for A, B and C. That is, we want to **unstack** this DataFrame.

Pivoting and Stacking

	date	variable	value
0	2000-01-03	A	1.234594
1	2000-01-04	A	0.661894
2	2000-01-05	A	0.810323
3	2000-01-03	B	-0.156366
4	2000-01-04	B	0.798020
5	2000-01-05	B	-0.360506
6	2000-01-03	C	0.375464
7	2000-01-04	C	0.413346
8	2000-01-05	C	-0.071480
9	2000-01-03	D	0.108641
10	2000-01-04	D	-0.738962
11	2000-01-05	D	0.460154

https://en.wikipedia.org/wiki/Pivot_table

The `pivot` method takes care of unstacking DataFrames. We supply indices for the new DataFrame, and tell it to turn the variable column in the old DataFrame into a set of column names in the unstacked one.

```
df.pivot(index='date',
         columns='variable',
         values='value')
```

variable	A	B	C	D
date				
2000-01-03	1.234594	-0.156366	0.375464	0.108641
2000-01-04	0.661894	0.798020	0.413346	-0.738962
2000-01-05	0.810323	-0.360506	-0.071480	0.460154

Pivoting and Stacking

Note: we could also construct the index set `tuples` using `itertools`.

```
tuples = list(zip(*[['bird','bird','goat','goat'],
                    ['x', 'y', 'x', 'y']]))
index = pd.MultiIndex.from_tuples(tuples,names=['animal','cond'])
df = pd.DataFrame(np.random.randn(4, 2),
                  index=index, columns=['A', 'B'])
df
```

		A	B
animal	**cond**		
bird	**x**	0.699732	-1.407296
	y	0.810211	1.249299
goat	**x**	-0.909280	0.184450
	y	-0.755891	-0.957222

How do we stack this? That is, how do we get a non-pivot version of this DataFrame? The answer is to use the DataFrame `stack` method.

Pivoting and Stacking

animal	cond	A	B
bird	x	0.699732	-1.407296
	y	0.810211	1.249299
goat	x	-0.909280	0.184450
	y	-0.755891	-0.957222

```
df.stack()
```

```
animal  cond
bird    x     A    0.699732
              B   -1.407296
        y     A    0.810211
              B    1.249299
goat    x     A   -0.909280
              B    0.184450
        y     A   -0.755891
              B   -0.957222
dtype: float64
```

```
s = df.stack()
s['bird']['x']['A']
```

```
0.6997320218227948
```

The DataFrame `stack` method makes a stacked version of the calling DataFrame. In the event that the resulting column index set is trivial, the result is a Series. Note that `df.stack()` no longer has columns A or B. The column labels A and B have become an extra index.

Pivoting and Stacking

```
columns = pd.MultiIndex.from_tuples(
    [('A', 'cat', 'long'), ('B', 'cat', 'long'),
     ('A', 'dog', 'short'), ('B', 'dog', 'short')],
    names=['cond', 'animal', 'hair_length'])
df = pd.DataFrame(np.random.randn(4, 4), columns=columns)
df
```

cond	A	B	A	B
animal	cat	cat	dog	dog
hair_length	long	long	short	short
0	-0.424446	-0.204965	-2.494808	1.278635
1	-0.710625	-0.801063	0.947879	0.763564
2	0.016435	0.701775	-0.577844	-1.315433
3	0.451242	0.886683	-0.864094	0.529257

Here is a more complicated example. Notice that the column labels have a three-level hierarchical structure.

There are multiple ways to stack this data. At one extreme, we could make all three levels into columns. At the other extreme, we could choose only one to make into a column.

Pivoting and Stacking

Stack only according to level 1 (i.e., the animal column index).

Missing animal x cond x hair_length conditions default to NaN.

cond	A	B	A	B
animal	cat	cat	dog	dog
hair_length	long	long	short	short
0	-0.424446	-0.204965	-2.494808	1.278635
1	-0.710625	-0.801063	0.947879	0.763564
2	0.016435	0.701775	-0.577844	-1.315433
3	0.451242	0.886683	-0.864094	0.529257

```
df.stack(level=1)
```

	cond	A		B	
	hair_length	long	short	long	short
	animal				
0	cat	-0.424446	NaN	-0.204965	NaN
	dog	NaN	-2.494808	NaN	1.278635
1	cat	-0.710625	NaN	-0.801063	NaN
	dog	NaN	0.947879	NaN	0.763564
2	cat	0.016435	NaN	0.701775	NaN
	dog	NaN	-0.577844	NaN	-1.315433
3	cat	0.451242	NaN	0.886683	NaN
	dog	NaN	-0.864094	NaN	0.529257

Pivoting and Stacking

cond	A	B	A	B
animal	cat	cat	dog	dog
hair_length	long	long	short	short
0	-0.424446	-0.204965	-2.494808	1.278635
1	-0.710625	-0.801063	0.947879	0.763564
2	0.016435	0.701775	-0.577844	-1.315433
3	0.451242	0.886683	-0.864094	0.529257

```
df.stack(level=[0,1,2])
```

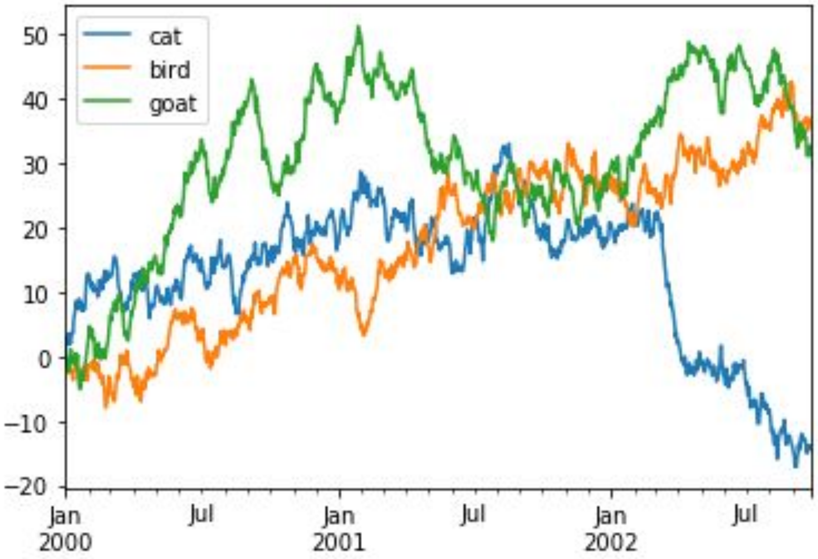
```
   cond  animal  hair_length
0  A     cat     long           -0.424446
         dog     short          -2.494808
   B     cat     long           -0.204965
         dog     short           1.278635
1  A     cat     long           -0.710625
         dog     short           0.947879
   B     cat     long           -0.801063
         dog     short           0.763564
2  A     cat     long            0.016435
         dog     short          -0.577844
   B     cat     long            0.701775
         dog     short          -1.315433
3  A     cat     long            0.451242
         dog     short          -0.864094
   B     cat     long            0.886683
         dog     short           0.529257
dtype: float64
```

Stacking across all three levels yields a Series, since there is no longer any column structure. This is often called **flattening** a table.

Notice that the NaN entries are not necessary here, since we have an entry in the Series only for entries of the original DataFrame.

Plotting DataFrames

```
df = pd.DataFrame(np.random.randn(1000, 3),
                  index=pd.date_range('1/1/2000', periods=1000),
                  columns=['cat','bird','goat'])
df = df.cumsum()
_ = df.plot()
```



`cumsum` gets partial sums, just like in `numpy`.

Note: this requires that you have imported `matplotlib`.

Note that legend is automatically populated and x-ticks are automatically date formatted.