STATS 507
Data Analysis in Python

Lecture 10: Basics of `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
- 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`

Anaconda:

```
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., `numpy` array, Python list, dict).

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

```
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.

```
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

```
1 s = pd.Series([2,3,5,7,11], index=['a','a','a','a','a'])
2 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 perform an operation that requires unique indices.

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

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

`pandas` Series

```
s
```

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

```
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.

```
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
```

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

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

```
---------------------------------------------------------------------------
KeyError
<ipython-input-48-a7df9b66ea8a>
----> 1 s['penguin']
```

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.

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

`pandas` Series

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.

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

```
s
```

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

```
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.
```

pandas Series

```
1 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.

```
1 s + 2*s
```

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

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

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

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

```
1 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

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

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

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

```
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 rows 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.

Rows 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

```
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.

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

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

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

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

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

	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

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

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

```
presidents
```

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

```
presidents
```

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

Select columns 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']]`.

```
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.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 columns 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

These expressions return Series objects.

```
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
```

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

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

These expressions return DataFrames.

```
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

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

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

```
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

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

```
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
```

Pandas Panels

Pandas has another data type, called a Panel

- Meant for representing panel data
  - https://en.wikipedia.org/wiki/Panel_data

Panel is deprecated, so I am not going to teach you about it

- But you should be aware that it exists, because it is mentioned in docs

https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Panel.html

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				

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

ame
py is

```
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).

Summarizing 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

`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.

```
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.

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 dispath 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

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

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

	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.

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

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

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

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

Element-wise function application

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

	A	B
0	cat	unicorn
1	dog	chupacabra
2	bird	pixie

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

	A	B
0	CAT	UNICORN
1	DOG	CHUPACABRA
2	BIRD	PIXIE

Tablewise Function Application

```
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

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

```
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

```
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

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.

```
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

Iterating over Series and DataFrames

```
1 s
```

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

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

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

```
fruit
animal
animal
fruit
fruit
```

```
1 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 a DataFrame gets an iterator over the column names.

```
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

```
1 for x in df.iterrows():
2     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.

```
1 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
```

4	-0.350299	1.958805	0.467778

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.