# 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)

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

2 import numpy as np
3 numbers = np.random.randn(5)
4 s = pd.Series(numbers)
5 s

0 -0.318743
1 0.807948

idx = ['a', 'b', 'c', 'd', 'e']

= pd.Series(numbers, index=idx)

import pandas as pd

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

3 -0.356014 4 1.542122 dtype: float64

-0.216362

Pandas tries to infer this data type automatically.

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

```
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.

```
1 d = {'dog':3.1415,'cat':42,'bird':0,'goat':1.618}
  2 s = pd.Series(d)
                                                      Can create a series from a
                                                      dictionary. Keys become indices.
bird
         0.0000
        42,0000
cat
dog
        3.1415
goat
        1.6180
dtype: float64
  1 inds = ['dog','cat','bird','goat','cthulu']
  2 s = pd.Series(d, index=inds)
  3 s
                                                     Index 'cthulu' doesn't appear in the
                                                     dictionary, so pandas assigns it NaN, the
dog
           3.1415
                                                     standard "missing data" symbol.
cat
         42,0000
bird
          0.0000
goat
          1.6180
cthulu
              NaN
dtype: float64
```

```
pandas Series
                                               Indexing works like you're used
                                               to and supports slices, but not
                                               negative indexing.
     s = pd.Series([2,3,5,7,11])
   2 s[0]
                                       This object has type np.int64
   1 s[1:3]
                                          This object is another
                                          pandas Series.
 dtype: int64
   1 s[-1]
                                               Traceback (most recent call last)
 KeyError
 <ipython-input-22-0e2107f91cbd> in <module>()
 ----> 1 s[-1]
```

```
1 s['a']

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

```
S
           3.1415
dog
          42.0000
cat
bird
           0.0000
           1.6180
goat
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.

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

update entries via their keys.

Series objects are dict-like, in that we can access and

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.

```
dog
           3.1415
          42.0000
cat
bird
           0.0000
           1.6180
goat
cthulu
               NaN
dtype: float64
  1 s['goat']
1.61800000000000001
    s['cthulu']=-1
  2 s
dog
           3.1415
```

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

1 s['penguin']

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

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

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

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.

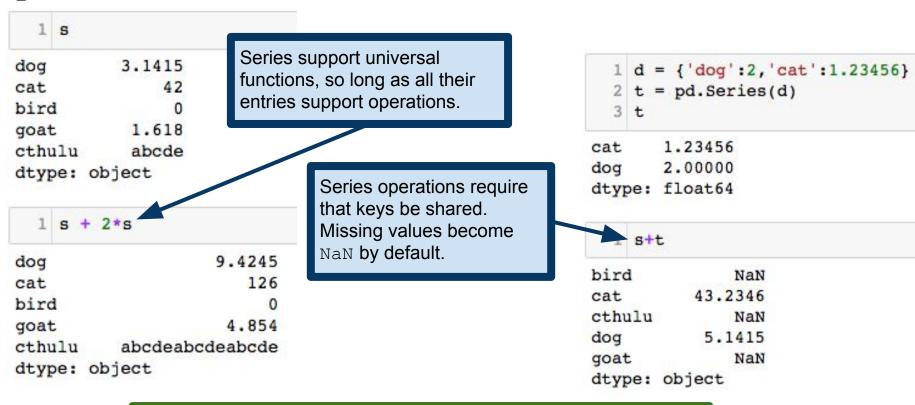
More information on indexing: <a href="https://pandas.pydata.org/pandas-d">https://pandas.pydata.org/pandas-d</a> ocs/stable/indexing.html

```
3.1415
dog
cat
         42.0000
bird
          0.0000
          1.6180
goat
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

ValueError: setting an array element with a sequence.



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

```
pandas Series
```

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

1 s

bird

Series have an optional name attribute.

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

0.0000

After it is set, name goat 1.6180

Name: aminals, dtype: float64

attribute can be changed with rename method.

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

```
bird 0.0000
cat 42.0000
dog 3.1415
goat 1.6180
```

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

Name: animals, dtype: float64

## Mapping and linking Series values

```
s = pd.Series(['dog', 'goat', 'skunk'])
                                          dog
                                         goat
                                        skunk
                                  dtype: object
Series map method works
analogously to Python's map
                                      s.map(lambda s:len(s))
function. Takes a function and
applies it to every entry.
                                  dtype: int64
```

# Mapping and linking Series values

```
s = pd.Series(['fruit', 'animal', 'animal', 'fruit', 'fruit'],
                   index=['apple','cat', 'goat', 'banana', 'kiwi'])
apple
          fruit
cat
       animal
goat animal
banana fruit
kiwi
       fruit
dtype: object
  1 t = pd.Series({'fruit':0, 'animal':1})
  2 s.map(t)
                                     Series map also allows us to change
apple
                                     values based on another Series. Here,
cat
                                     we're changing the fruit/animal category
goat
                                     labels to binary labels.
banana
kiwi
dtype: int64
```

#### 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.

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

Rows that are unspecified for a given column receive NaN.

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.

**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: <a href="https://pandas.pydata.org/pandas-docs/stable/generated/pandas.">https://pandas.pydata.org/pandas-docs/stable/generated/pandas.</a>
<a href="DataFrame.html">DataFrame.html</a>

## pandas DataFrames: creating DataFrames

Dictionary has 4 keys, so 4 columns.

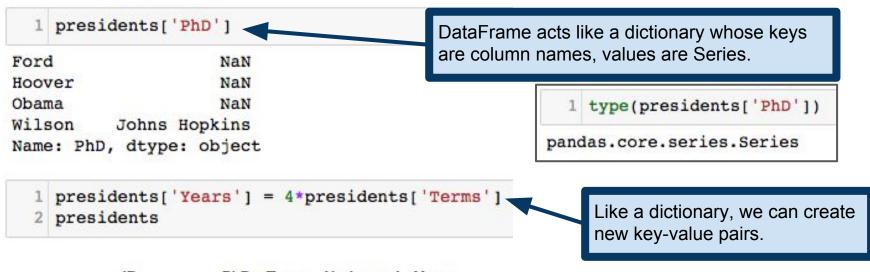
	JD	PND	ierms	Undergrad
Ford	Yale	NaN	1	UMich
Hoover	NaN	NaN	1	Stanford
Obama	Harvard	NaN	2	Columbia
Wilson	NaN	Johns Hopkins	2	Princeton

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	Row and column names accessible as the index and column attributes, respectively, of the DataFrame.
1 pr	esident	s.columns			respondition, or the Battar raine.
			Terms	', u'Underg	rad'], dtype='object')

# 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

**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]

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: <a href="https://pandas.pydata.org/pandas-d">https://pandas.pydata.org/pandas-d</a> ocs/stable/generated/pandas.DataFr ame.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

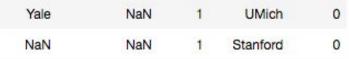
	JD	PhD	Terms	Undergrad	Nobels	Years	Fields Medals
Ford	Yale	NaN	1	UMich	0	4	0
Hoover	NaN	NaN	1	Stanford	0	4	0
Obama	Harvard	NaN	2	Columbia	1	8	0
Wilson	NaN	Johns Hopkins	2	Princeton	1	8	0

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

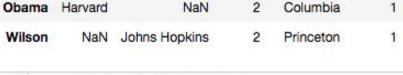
del presidents['Years']
presidents

	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

#### Indexing and selection PhD Terms Undergrad Nobels JD **UMich** Ford Yale NaN NaN NaN Stanford Hoover



NaN Columbia



Obama Wilson

Ford

Hoover

NaN Harvard NaN Name: JD, dtype: object

Yale

1 presidents[1:3]

1 presidents['JD']

JD Harvard PhD NaN Terms Undergrad Columbia

1 presidents.loc['Obama']

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

Undergrad Nobels Stanford 2

Nobels Name: Obama, dtype: object

Columbia

1 presidents.iloc[1]

PhD

NaN NaN

JD

1 presidents[presidents['Terms']<2]</pre>

0

JD PhD Undergrad Terms

Nobels 0

Terms Undergrad Nobels

Stanford

Name: Hoover, dtype: object

Ford Yale Hoover NaN

NaN NaN

UMich 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

Select columns by their names.

JD Harvard PhD NaN Terms

1 presidents

Undergrad

Nobels

Nobels

Columbia

Name: Obama, dtype: object

1 presidents.iloc[1]

JD NaN PhD NaN Terms Undergrad Stanford

1 presidents['JD'] Ford Yale Hoover NaN Obama Harvard Wilson NaN Name: JD, dtype: object 1 presidents[1:3]

NaN

Hoover Obama

Harvard 1 presidents[presidents['Terms']<2]</pre>

JD

Yale

NaN

PhD

PhD

NaN

NaN

Terms Undergrad Nobels **UMich** 

Nobels

0

0

0

Name: Hoover, dtype: object

Ford

Hoover

NaN

NaN

Stanford

Terms Undergrad

Stanford

Columbia

## Indexing and selection

	JD		Terms	Undergrad	Nobel
Ford	Yale Yale	NaN	1	UMich	(
Hoove	NaN	NaN	1	Stanford	
Obama	Harvard	NaN	2	Columbia	
Wilson	NaN	Johns Hopkins	2	Princeton	

1 presidents Select columns by their numerical JD indices (again 0-indexed). This PhD supports slices. Terms Undergrad Сотишрта Nobels Note: one can also select slices Name: Obama, d with lists of column names, e.g., presidents[['JD','PhD']]. 1 presidents JD NaN PhD NaN Terms Undergrad Stanford Nobels

Name: Hoover, dtype: object

1 presidents['JD'] Yale Ford Hoover NaN Obama Harvard

Mary at other			:	
resident	ts[1:	3]		
JD	PhD	Terms	Undergrad	١
r NaN	NaN	1	Stanford	
a Harvard	NaN	2	Columbia	
	JD, dty president JD er NaN	JD, dtype: presidents[1:     JD PhD     NaN NaN	JD, dtype: object presidents[1:3]  JD PhD Terms er NaN NaN 1	JD, dtype: object  presidents[1:3]  JD PhD Terms Undergrad  er NaN NaN 1 Stanford

			JD	PhD	Terr	ns	Undergra	ad	Nobel
Hoo	ver	Na	aN	NaN		1	Stanfo	rd	
Oba	ma	Harva	ard	NaN		2	Columb	oia	
1	pr	eside	ent	s[pr	esi	der	nts['Te	rm	s']<2
		JD	Ph	D Te	rme	110	ndergrad	No	hele
				CO 05-67	TITIO	U	lucigiau	140	, DCI3
Fo	ord	Yale	Na		1	0,	UMich	140	0

# Indoving and coloction

		JD	PhD	Terms	Undergrad	Nobels
F	ord	Yale	NaN	1	UMich	0
Hoo	ver	NaN	NaN	1	Stanford	0
Oba	ma	Harvard	NaN	2	Columbia	1
Wils	on	NaN	Johns Hopkins	2	Princeton	1
1	presidents.loc['Obama']					
JD.	Harvard					
PhD			NaN			

NaN

Terms Undergrad Columbia Nobels

Name: Obama, dtype: object

1 presidents.iloc[1] JD PhD

Terms

NaN NaN Undergrad Stanford Nobels

Name: Hoover, dtype: object

Select columns by Boolean expression. Hoover

Ford Hoover

Obama

Wilson

NaN NaN Obama Harvard NaN

Yale

NaN

Ford

Hoover

JD PhD

NaN

NaN

presidents[presidents['Terms']<2]</pre>

Name: JD, dtype: object

1 presidents['JD']

Yale

Harvard

NaN

NaN

1 presidents[1:3]

2

PhD Terms Undergrad Nobels Stanford

Columbia

0

Terms Undergrad Nobels

0

0

**UMich** Stanford



#### Indexing and selection PhD Terms Undergrad Nobels JD

**UMich** Ford Yale NaN 0 NaN NaN Stanford 0 Hoover 1 Obama Harvard NaN Columbia Wilson NaN Johns Hopkins Princeton

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

1 presidents[1:3]

NaN

Harvard

presidents['JD']

presidents.loc['Obama'] These expressions return Series objects. Harvard NaN

Hoover Obama

PhD Terms NaN

NaN

Stanford 0 Columbia

Nobels

0

Undergrad

1 presidents[presidents['Terms']<2]</pre>

2

Nobels Name: Obama, dtype: object presidents.iloc[1] JD NaN PhD NaN Terms Undergrad Stanford Nobels Name: Hoover, dtype: object

Columbia

JD

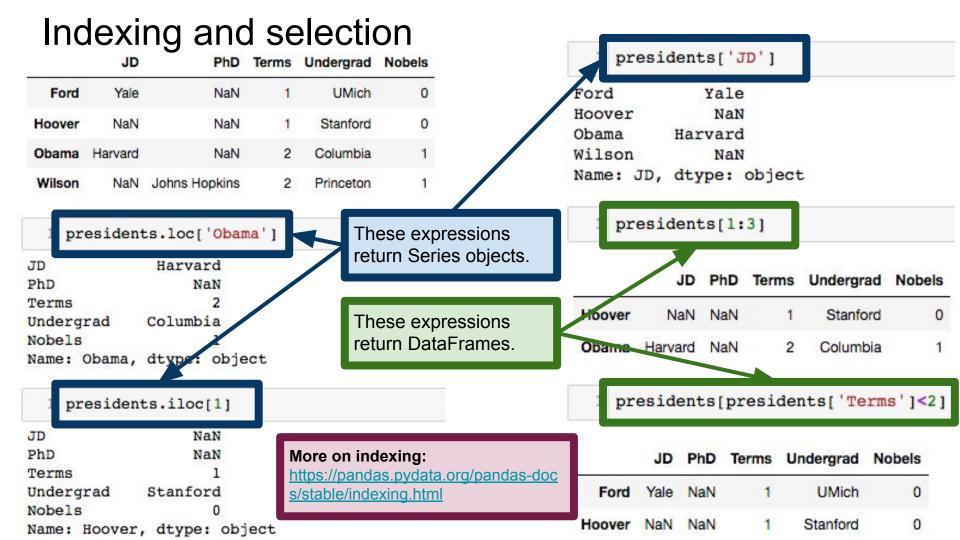
PhD

Terms

Undergrad

JD PhD Undergrad Nobels Terms Yale **UMich** Ford NaN 0

Hoover NaN NaN Stanford



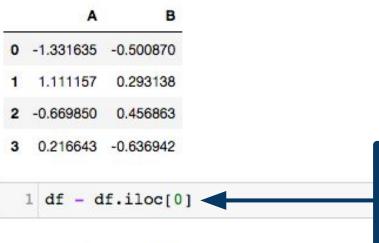
```
1 df1 = pd.DataFrame(np.random.randn(8, 4), columns=['A', 'B', 'C', 'D'])
2 df2 = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])
3 df1+df2
```

	A	В	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.

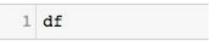
```
1 df = pd.DataFrame(np.random.randn(4, 2), columns=['A', 'B'])
2 df
```

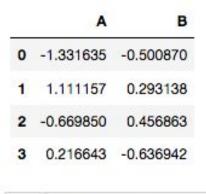


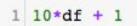
By default, Series are aligned to DataFrames via row-wise broadcasting.

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
```







A B

0 -12.316346 -4.008702

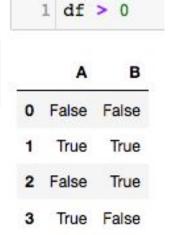
1 12.111569 3.931385

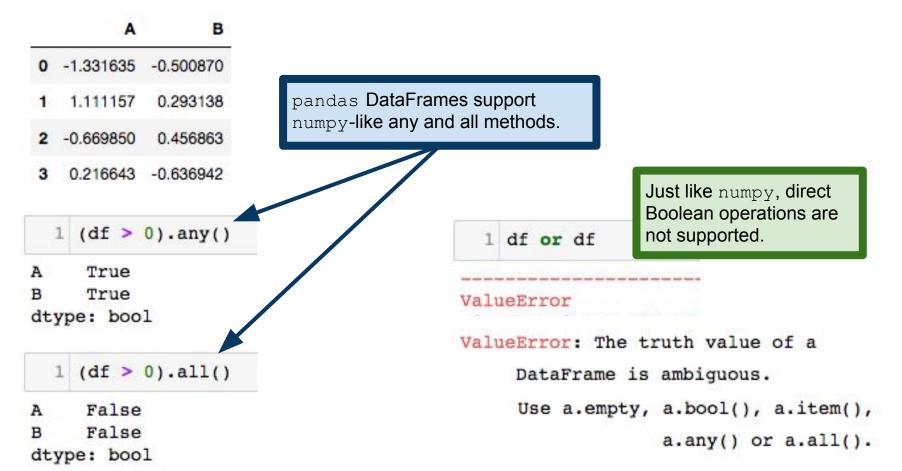
2 -5.698497 5.568633

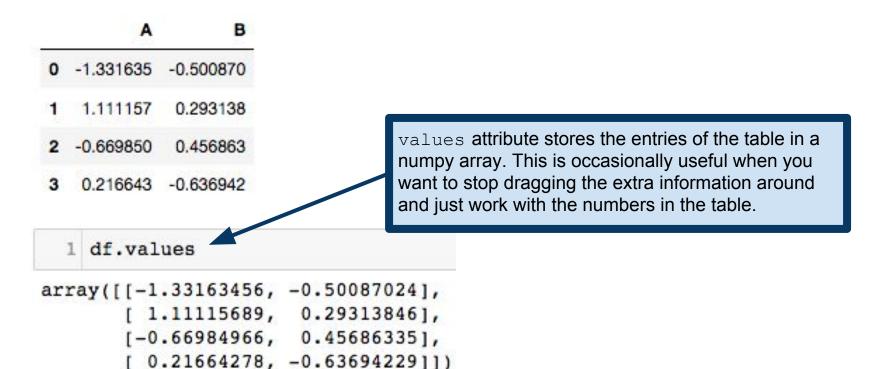
3 3.166428 -5.369423

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.









A B
0 -1.331635 -0.500870

df

1 1.111157 0.293138

**2** -0.669850 0.456863

3 0.216643 -0.636942

DataFrames support entrywise multiplication. The  $\ensuremath{\mathbb{T}}$  attribute is the transpose of the DataFrame.

df.T\*df.T

0

1.773251 1.23467 0.448699 0.04693

**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.

A B

df.T.dot(df)

A 3.503553 0.548680

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

# Removing NaNs

-9.422331 1,100197 8.034010 NaN -1.5201405.655382 -1.692761 NaN 0.399654 10.058568 0.502007 NaN -4.070947 2.237868 10.530079 NaN 1.603739 8.255591 1.892258 NaN 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.

-9.422331 1.100197 -1.520140 5.655382 0.399654 10.058568 3 -4.070947 2.237868 1.603739 8.255591 1.123450 3.141590 df.dropna(axis=1, how='all')

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

0 -9.422331 1.100197 8.034010 -1.5201405.655382 -1.69276110.058568 0.502007 0.399654 -4.070947 2.237868 10.530079 1.603739 8.255591 1.892258 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
hinary	MS Excel	read_excel	to_excel
ala a fila I/O	) in largely similar to D 1 + -1-1 -	read hdf	to hdf
das IIIe I/	O is largely similar to R read.table	Toda_Hai	10_1101
	nctions, so I'll leave it to you to read the	read_feather	to_feather
similar fur			
similar fur	nctions, so I'll leave it to you to read the	read_feather	to_feather to_parquet
similar fur das docui	nctions, so I'll leave it to you to read the mentation as needed.	read_feather read_parquet	to_feather to_parquet
similar fur das docui binary	nctions, so I'll leave it to you to read the mentation as needed.  Msgpack	read_feather read_parquet read_msgpack	to_feather to_parquet to_msgpack
similar fur das docui binary binary	nctions, so I'll leave it to you to read the mentation as needed.  Msgpack Stata	read_feather read_parquet read_msgpack read_stata	to_feather to_parquet to_msgpack
similar fur das docui binary binary binary	mentation as needed.  Msgpack Stata SAS	read_feather read_parquet read_msgpack read_stata read_sas	to_feather to_parquet to_msgpack to_stata

Table credit: <a href="https://pandas.pydata.org/pandas-docs/stable/io.html">https://pandas.pydata.org/pandas-docs/stable/io.html</a>

### **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()
 lass '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
         21699 non-null object
team
         21634 non-null object
lq
         21699 non-null int64
ab
         21699 non-null int.64
         21699 non-null int64
         21699 non-null int64
X2b
         21699 non-null int.64
x3b
         21699 non-null int64
hr
         21699 non-null int64
rbi
         21687 non-null float64
         21449 non-null float64
sb
         17174 non-null float64
CS
bb
         21699 non-null int.64
         20394 non-null float64
SO
ibb
         14171 non-null float64
hbp
         21322 non-null float64
sh
         20739 non-null float64
sf
         14309 non-null float64
qidp
        16427 non-null float64
dtypes: float64(9), int64(10), object(3)
memory usage: 3.8+ MB
```

### **Summarizing DataFrames**

1 baseball.head()

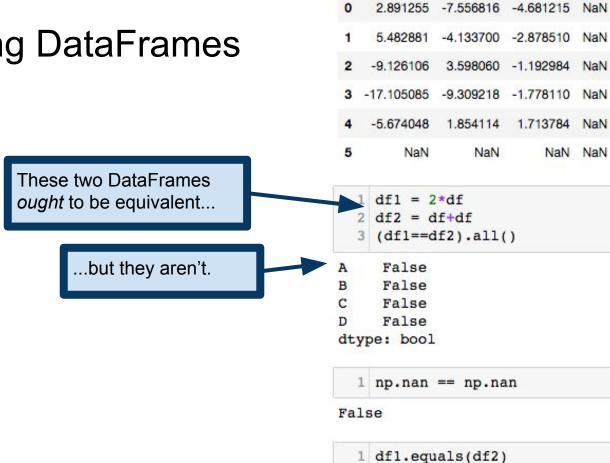
head() method displays just the first few rows of the DataFrame (5 by default; change this by supplying an argument). tail() displays the last few rows.

	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 x 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



В

C

D

True

### **Comparing DataFrames**

These two DataFrames ought to be equivalent...

...but they aren't.

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

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

```
2.891255 -7.556816 -4.681215 NaN
    5.482881
             -4.133700 -2.878510 NaN
    -9.126106
             3.598060 -1.192984 NaN
   -17.105085 -9.309218 -1.778110 NaN
    -5.674048
             1.854114
                       1.713784 NaN
        NaN
                 NaN
                           NaN
                               NaN
    df1 = 2*df
    df2 = df+df
    (df1==df2).all()
     False
     False
     False
     False
dtype: bool
    np.nan == np.nan
False
    dfl.equals(df2)
```

True

### **Comparing DataFrames**

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.

```
        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

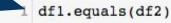
        5
        NaN
        NaN
        NaN
        NaN
```

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

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

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

False



True

### **Pandas Panels**

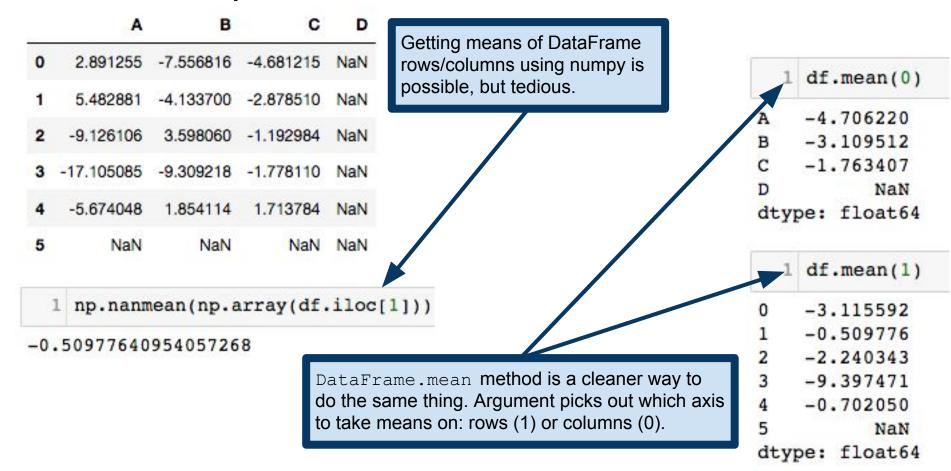
Pandas has another data type, called a Panel
Meant for representing panel data
<a href="https://en.wikipedia.org/wiki/Panel\_data">https://en.wikipedia.org/wiki/Panel\_data</a>

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

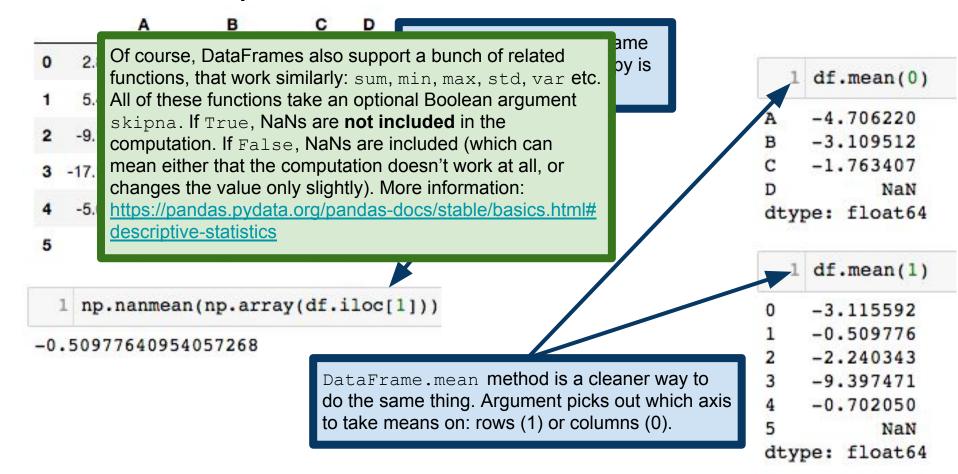
But you should be aware that it exists, because it is mentioned in docs

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

### Statistical Operations on DataFrames



### Statistical Operations on DataFrames



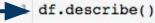
## **Summarizing DataFrames**

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.

### **Details and optional arguments:**

https://pandas.pydata.org/pandas-docs/stable/basics.ht ml#summarizing-data-describe

	5740	С	D
2.891255	-7.556816	-4.681215	NaN
5.482881	-4.133700	-2.878510	NaN
-9.126106	3.598060	-1.192984	NaN
17.105085	-9.309218	-1.778110	NaN
-5.674048	1.854114	1.713784	NaN
NaN	NaN	NaN	NaN
	5.482881 -9.126106 17.105085 -5.674048	5.482881 -4.133700 -9.126106 3.598060 17.105085 -9.309218 -5.674048 1.854114	-9.126106 3.598060 -1.192984 17.105085 -9.309218 -1.778110 -5.674048 1.854114 1.713784



	A	В	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

		A	В	С	D
Row- and column-wise functions: apply()	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.	A B	-0.293 0.465	070	an)	
	C D dty	-0.456 ppe: flo	NaN		
Axis argument is 0 by default (column-wise). Change to 1 for row-wise application.	0 1 2 3 4 5	0.885 0.186 -0.402 -1.102 -0.042	546 101 192	an, axis	=1)
	37792	pe: flo			

		A	В	(	D
Row- and column-wise functions: apply()	0	1.284355	1.073402	0.29757	5 NaN
	1	-0.791592	0.841969	0.509262	2 NaN
	2	-0.657900	-2.184139	1.63573	6 NaN
	3	-1.897574	0.502787	-1.911790	) NaN
	4	0.592821	2.091333	-2.813032	2 NaN
Numpy ufuncs take vectors and spit out vectors, so using df.apply() to apply a	5	NaN	NaN	Nal	NaN
ufunc to every row or column in effect ends up applying the ufunc to every element.		df.app	oly(np.e	xp)	
		A	В	С	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







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
7.759325 13.287581 4.22553
```

4.160428

5.045253

4.350433

5.640946

8.269566

4.003679

8.471805

14.612481

def quadratic(x, a, b, c=1):

return a\*x\*\*2 + b\*x + c

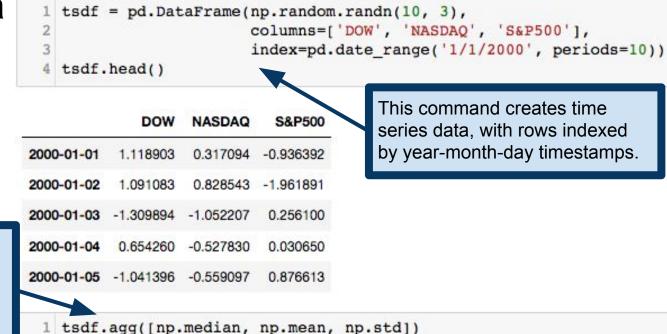
df.apply(quadratic, args=(1,2), c=5)

4.225538 4.647050 10.017612 4.035691

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



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.

 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

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

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.

	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

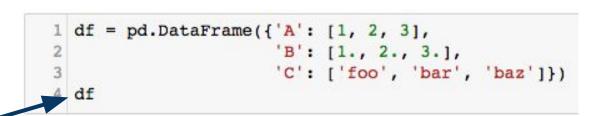
```
NASDAQ 0.230327
S&P500 0.876613
DOW 0.391512
dtype: float64
```

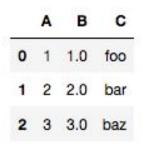
### Aggregating data

df contains mixed data types.

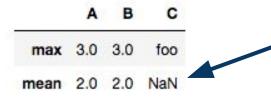
agg (and similarly apply) will only try to apply these functions on the columns of types supported by those functions.

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





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



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

### Element-wise function application

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

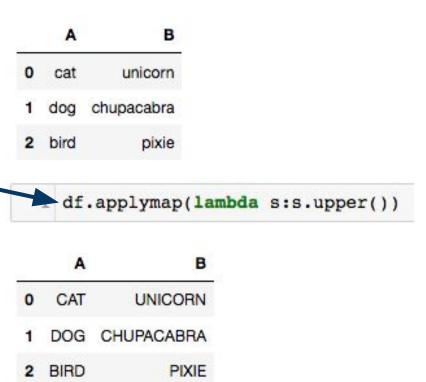
```
0 cat unicorn
1 dog chupacabra
2 bird pixie

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

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

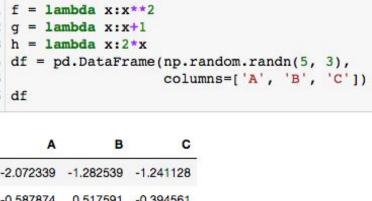
### Element-wise function application

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



### **Tablewise Function Application**

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





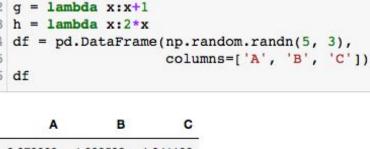
	A	В	C
0	10.589182	5.289812	5.080798
1	2.691193	2.535802	2.311357
2	2.054078	6.207308	3.902906
3	4.955251	2.901113	2.310556
4	2.245419	9.673833	2.437633

### **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.

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

	A	В	С
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





1 h(g(f(df)))

= lambda x:x\*\*2

# 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

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

1 for x in s: 2 print x

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

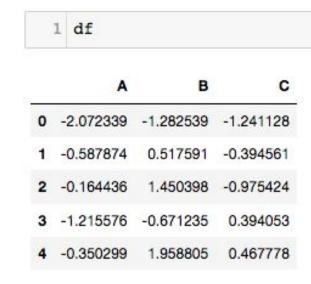
Iterating over a DataFrame gets an iterator over the column names.

1 df -2.072339 -1.282539 -1.241128 -0.5878740.517591 -0.394561 2 -0.164436 1.450398 -0.975424 -1.215576 -0.671235 0.394053 4 -0.350299 1.958805 0.467778 for x in df: print x A В

```
for x in df.iteritems():
        print(x)
 'A', 0 -2.072339
    -0.587874
   -0.164436
   -1.215576
    -0.350299
Name: A, dtype: float64)
 B', 0 -1.282539
     0.517591
     1.450398
   -0.671235
    1.958805
Name: B, dtype: float64)
('C', 0 -1.241128
    -0.394561
   -0.975424
    0.394053
     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.



Name: 4, dtype: float64)

```
1 for x in df.iterrows():
        print(x)
                                DataFrame iterrows() returns an
(0, A -2.072339
                                iterator over the rows of the DataFrame
    -1.282539
    -1.241128
                                as (index, Series) pairs.
Name: 0, dtype: float64)
(1, A -0.587874
     0.517591
                                                1 df
    -0.394561
Name: 1, dtype: float64)
(2, A -0.164436
     1.450398
    -0.975424
                                              0 -2.072339 -1.282539 -1.241128
Name: 2, dtype: float64)
                                                          0.517591
                                                 -0.587874
                                                                  -0.394561
(3, A -1.215576
    -0.671235
                                                         1.450398 -0.975424
                                              2 -0.164436
     0.394053
                                              3 -1.215576 -0.671235
Name: 3, dtype: float64)
                                                                   0.394053
(4, A -0.350299
                                              4 -0.350299
                                                          1.958805
                                                                  0.467778
     1.958805
     0.467778
```

```
1 for x in df.iterrows():
        print(x)
       -2.072339
(0, A
   -1.282539
   -1.241128
Name: 0, dtype: float64)
(1, A
       -0.587874
    0.517591
   -0.394561
Name: 1, dtype: float64)
(2, A -0.164436
     1,450398
   -0.975424
Name: 2, dtype: float64)
(3, A -1,215576
    -0.671235
     0.394053
Name: 3, dtype: float64)
(4, A -0.350299
     1.958805
     0.467778
Name: 4, dtype: float64)
```

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

1 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