Numpy, Pandas,
&
Dow Jones Industrial Index

Git Update

- First time

```
git clone https://github.com/philmui/datascience2016fall
```

- All subsequent times (if you are not retaining any of your changes):

```
git clean --fd
git reset --hard
git fetch --all
```

Groups

- Form groups of 4 students
- Help everyone to be able to ”`git fetch --all`” for their individual course repo
- Help everyone in your group to run Lecture 02’s “`tweetering.py`” on all real-time tweets containing either:
  - “`trump`”
  - “`clinton`”

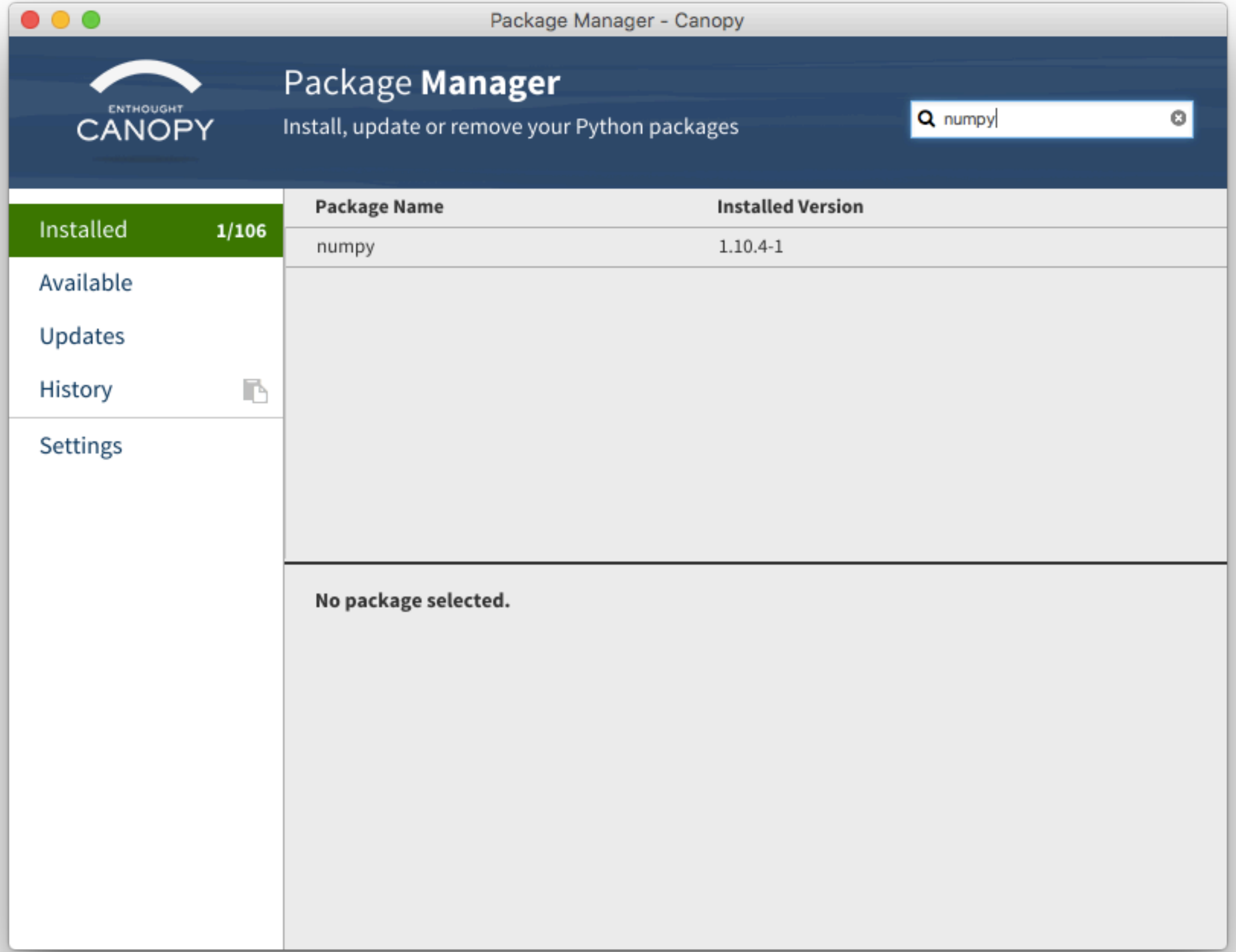
Agenda

- Numpy
- Pandas
- Finance
- Group Exercise
- Assignment

Agenda

- **Numpy**
- Pandas
- Finance
- Group Exercise
- Assignment

NumPy



Package Manager

Install, update or remove your Python packages

Package Name	Installed Version
numpy	1.10.4-1

No package selected.

numpy module

Items are all the same type.

Contiguous data storage in memory of items.

Considerably faster than lists.

Class with data and methods (subroutines).

numpy module

```
import numpy

dir()
dir(numpy)
help(numpy)
help(numpy.ndarray)      # class
help(numpy.array)           # built-in function
```

numpy module

```
import numpy
dir(numpy)
help(numpy.zeros)                tuple

a = numpy.zeros( (3,5) )
                        # create 3 rows, 5 columns
        [   [ 0., 0., 0., 0., 0. ],
            [ 0., 0., 0., 0., 0. ],
            [ 0., 0., 0., 0., 0. ] ]
                        # default type is float64
```

numpy Array Access

Access order corresponding to printed order:

[row] [column] index starts with 0

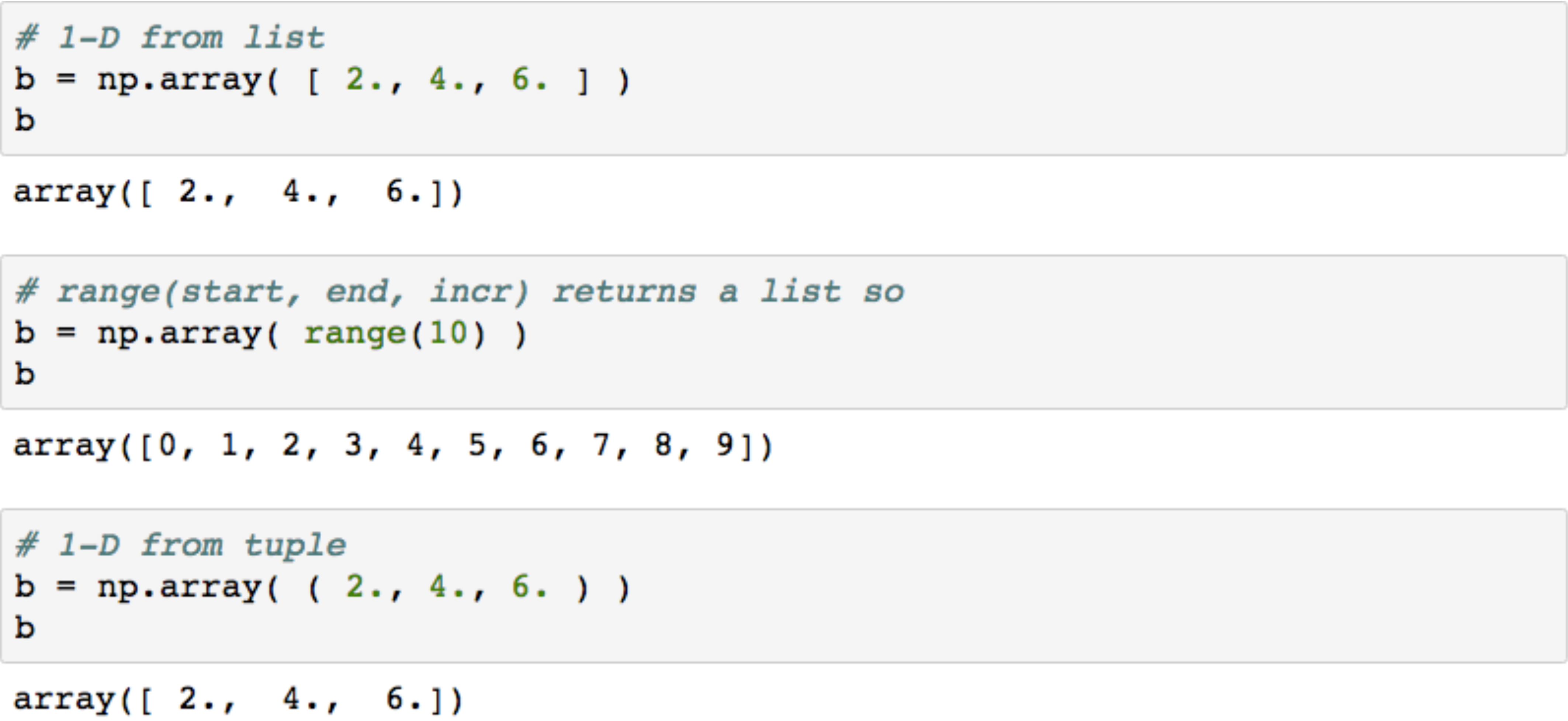
```
a[0][2] = 5
```

```
[      [ 0.,  0.,  5.,  0.,  0. ],
       [ 0.,  0.,  0.,  0.,  0. ],
       [ 0.,  0.,  0.,  0.,  0. ]  ]
```

NumPy arrays versus Python lists

- Python lists: Very general
  - `a = [1,2]`
  - `b = [[1,2],[3,4]]`
  - `c = [[1,2, ’duh’],[3,[4]]]`
- NumPy arrays:
  - `x = array([1,2])`
  - `y = array([[1,2],[3,4]])`
  - All rows must have same length, etc.
  - All entries must have same data-type, e.g. all real or all complex

Create 1-D Array



```
# 1-D from list
b = np.array( [ 2., 4., 6. ] )
b
```

```
array([ 2.,  4.,  6.])
```

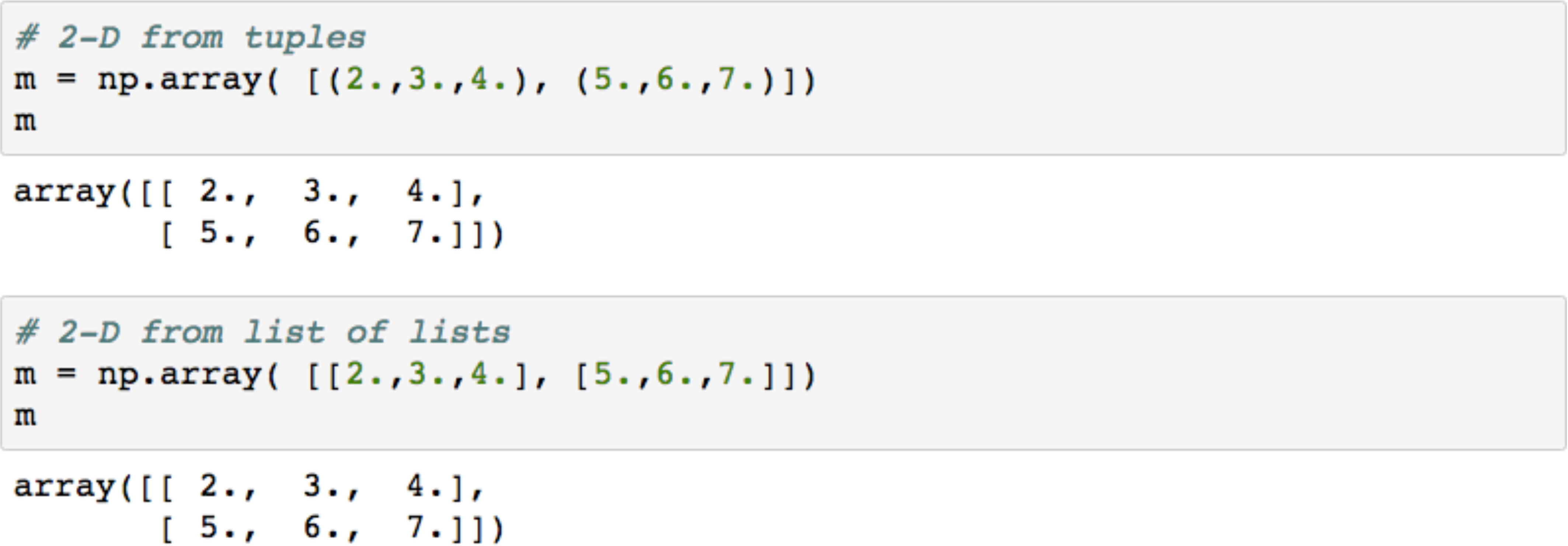
```
# range(start, end, incr) returns a list so
b = np.array( range(10) )
b
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
# 1-D from tuple
b = np.array( ( 2., 4., 6. ) )
b
```

```
array([ 2.,  4.,  6.])
```

Create 2-D Matrix



```
# 2-D from tuples
m = np.array( [(2.,3.,4.), (5.,6.,7.)])
m
```

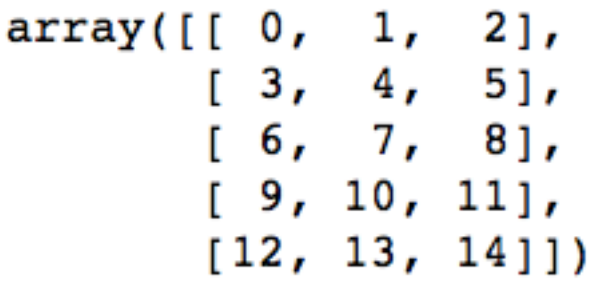
```
array([[ 2.,  3.,  4.],
       [ 5.,  6.,  7.]])
```

```
# 2-D from list of lists
m = np.array( [[2.,3.,4.], [5.,6.,7.]])
m
```

```
array([[ 2.,  3.,  4.],
       [ 5.,  6.,  7.]])
```

Group Exercise

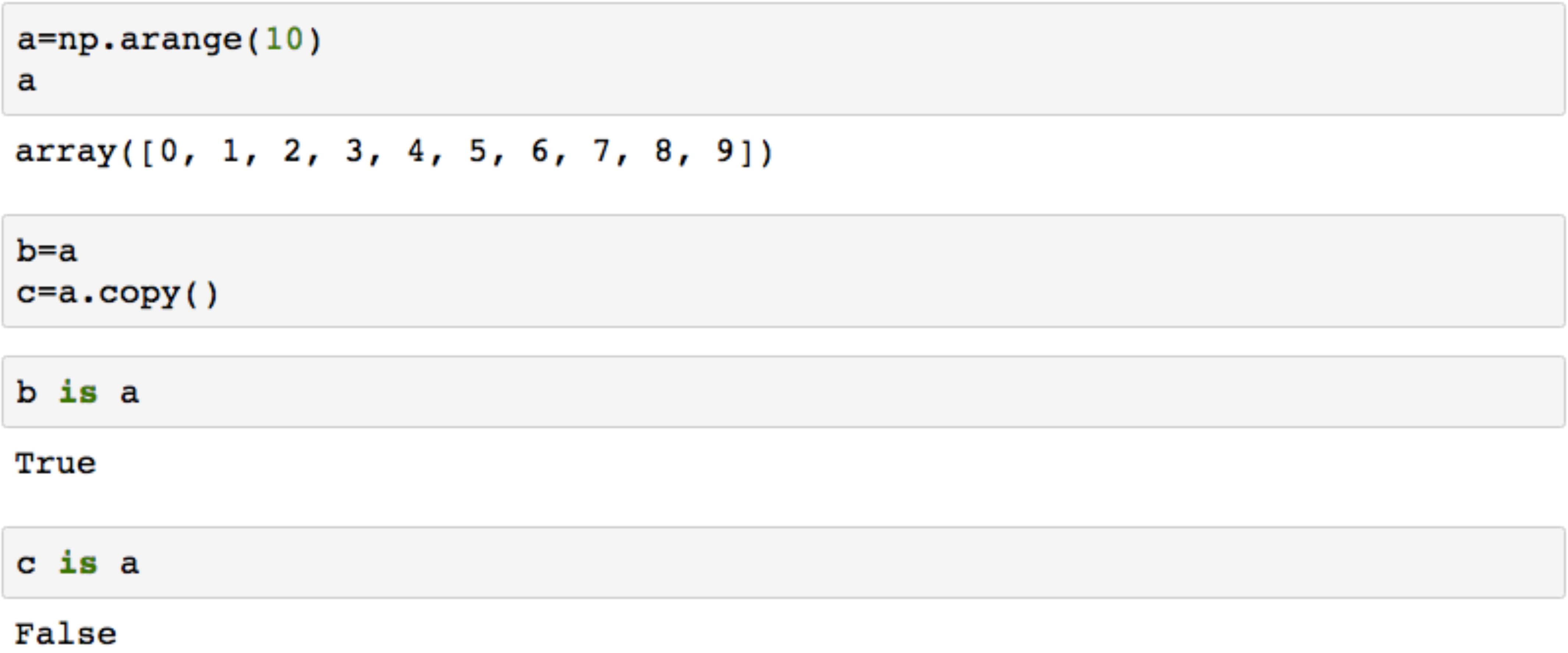
Create a (5, 3) 2-d array / matrix with Numpy that looks like the following:



```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Challenge: do it in 1 line

Pointer vs. Deep Copy



```
a=np.arange(10)
a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
b=a
c=a.copy()
```

```
b is a
```

```
True
```

```
c is a
```

```
False
```

Pointer vs. Deep Copy

```
a = numpy.zeros( (3, 3) )
b = a   # b is a pointer to a
c = a.copy()  # c is a new array

b is a   # True
c is a   # False
```

Views

base

Array Arithmetic

```
a = numpy.array( range(10, 20) )

a + 5
a – 3
a * 5
a / 3.14
a.sum()
a > 15
    (a > 15).sum()
```

Array Arithmetic by Index

```
a = numpy.array( range(10) )
b = numpy.array( range(0, 1000, 100) )

a + b           # a[0] + b[0], a[1] + b[1] ...
a – b
a * b           # not row, column matrix product
a / b
```

The 2 arrays must be the same shape.

Row, Column Matrix Product

```
c = numpy.dot(a, b)
```

Dot product of 2 arrays.
Matrix multiplication for 2D arrays.

Cross Product

```
zA = numpy.cross(xA, yA)
```

Note: we have been using *numpy.* functions

Array Shape

```
a = numpy.linspace(2, 32, 16)

a = a.reshape(4, 4) # ndarray . method

a.shape        # ndarray attribute    tuple (4, 4)

a = numpy.linspace(2,32,16).reshape(8,2)
```

Array Diagonals

```
a = numpy.linspace(1, 64, 64)

a = a.reshape(8, 8)

numpy.triu(a)     # upper triangle
numpy.tril(a)     # lower triangle

numpy.diag(a)     # main diagonal
numpy.diag(a, 1)  # 1 above
numpy.diag(a, -1) # 1 below
```

Array Data Types

numpy.float64 is the default type

float32

int8, int16, int32, int64, uint8, uint16, uint32, uint64

complex64, complex128

bool - True or False

a.dtype shows type of data in array

```
>>> help(numpy.ndarray)
```

Parameters

Attributes

Multi-Dimensional Indexing

```
a = numpy.array( range(12) )
a = a.reshape(2,6)      # 2 rows, 6 columns
```

a[1][5] contains 11

a[1, 5] is equivalent, more efficient

Array Slicing

```
a = numpy.array(range(0, 100, 10))
```

Array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90])

a[2:4] contains 20, 30

a[-4 : -1] contains 60, 70, 80

Slicing returns ndarray

Array Slicing

```
a = numpy.array(range(64)).reshape(8,8)
```

a[3, 4] contains 28

```
asub = a[3:5,  4:6]
```

Very useful for looking at data & debugging.

```
a[:,2]        # all rows, column 2
a[3, 2:5]     # row 3, columns 2 and 3 and 4
```

Array Stuff

```
a.T
a.min()
a.max()
a.round()
a.var()
a.std()
```

Organize Arrays

Make a list of arrays named a, b, and c:

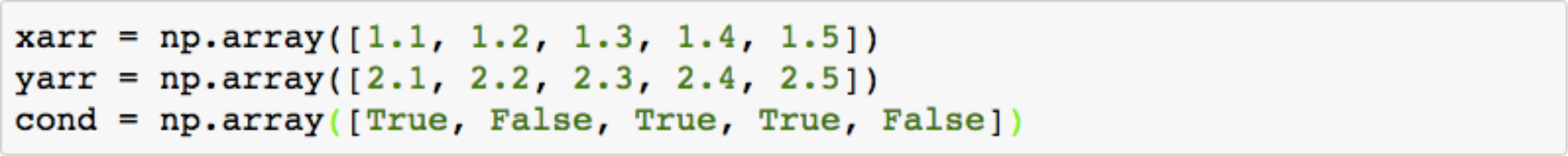
```
w = [ a, b, c]

len(w)          # length of list is 3

w[1].max()      # use array method
```

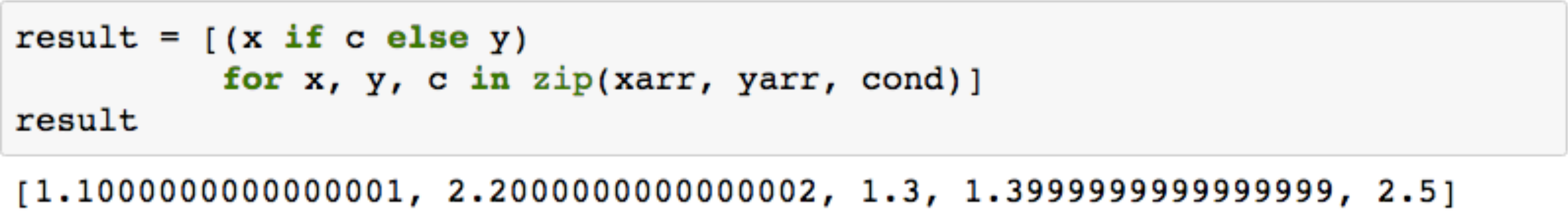
Conditional Logic with NumPy

Consider these arrays:



```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])
```

Use native “list comprehension” from Python:

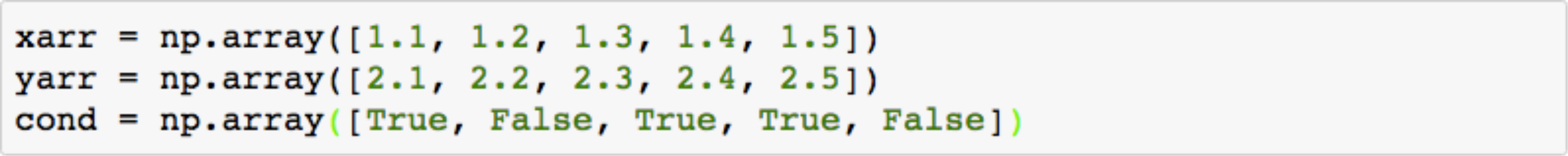


```
result = [(x if c else y)
          for x, y, c in zip(xarr, yarr, cond)]
result
```

```
[1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

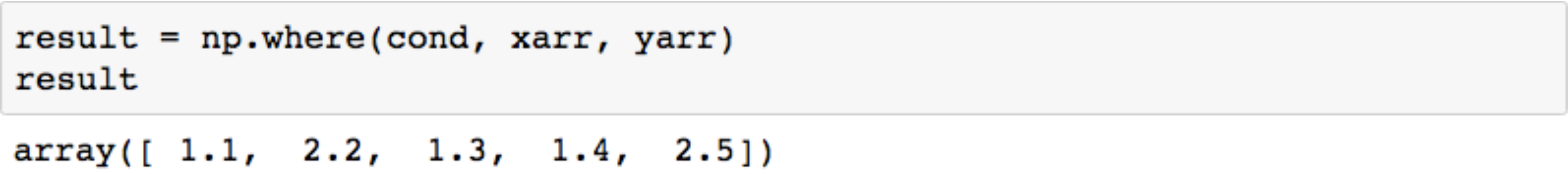
Conditional Logic with NumPy

Consider these arrays:



```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])
```

Use NumPy conditional logic:

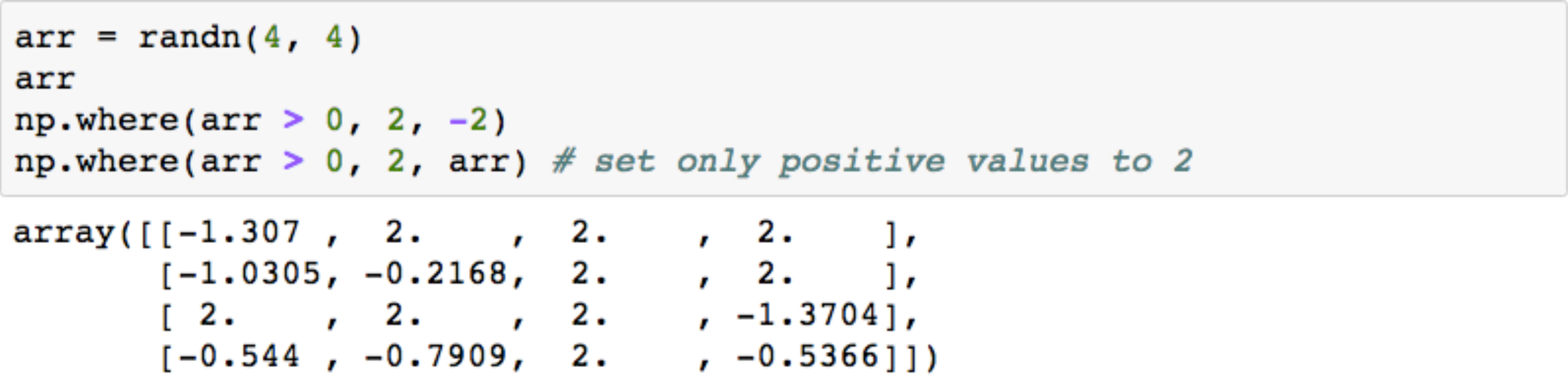


```
result = np.where(cond, xarr, yarr)
result
```

```
array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

Why Conditional Logic with NumPy?

Consider these arrays:



```
arr = randn(4, 4)
arr
np.where(arr > 0, 2, -2)
np.where(arr > 0, 2, arr) # set only positive values to 2
```

```
array([[-1.307 ,  2.    ,  2.    ,  2.    ],
       [-1.0305, -0.2168,  2.    ,  2.    ],
       [ 2.    ,  2.    ,  2.    , -1.3704],
       [-0.544 , -0.7909,  2.    , -0.5366]])
```

(1) Works with vectors / arrays / list by default

(2) Fast

EXERCISE

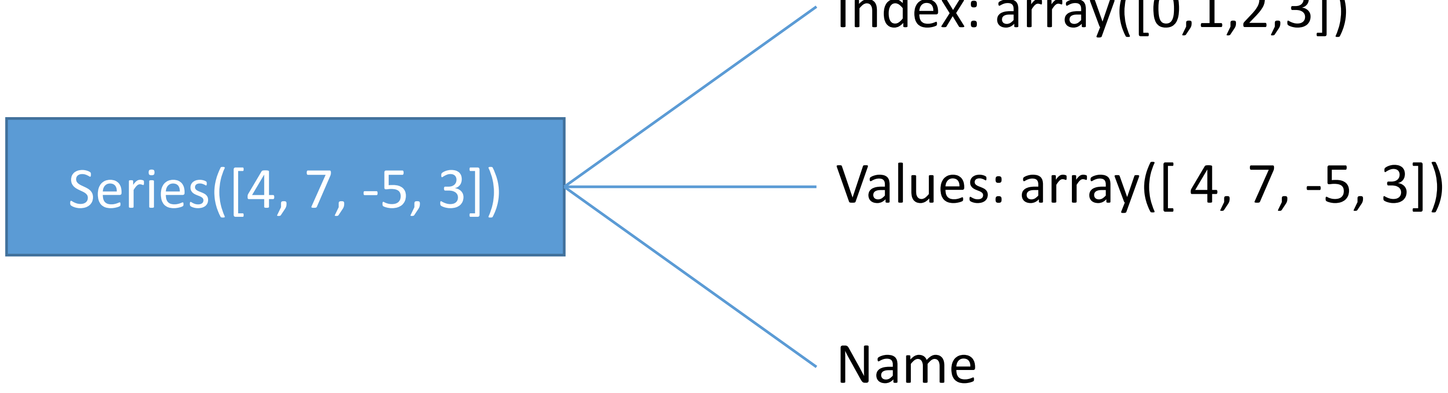
Pandas

Agenda

- Numpy
- **Pandas**
- Finance
- Group Exercise
- Assignment

Pandas

Series : pandas 1-D vectors



Series([4, 7, -5, 3])

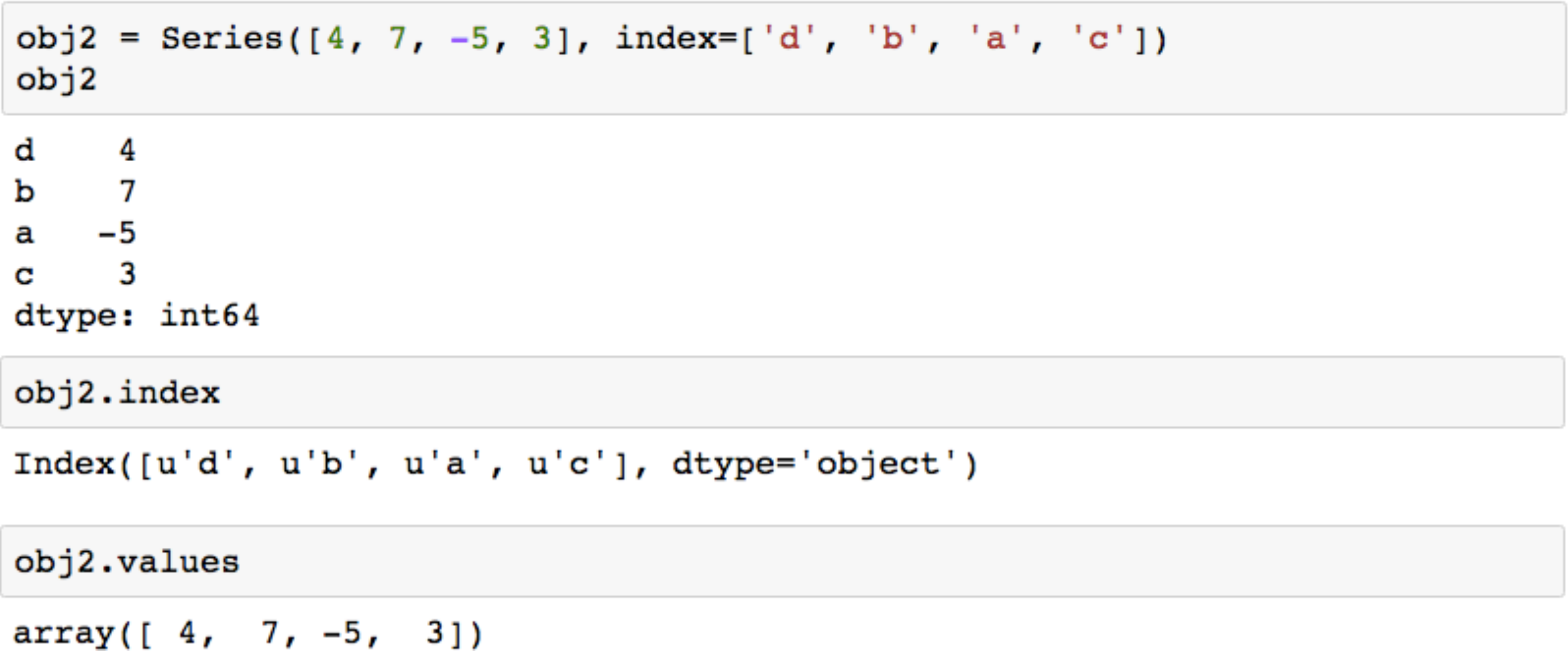
Index: array([0,1,2,3])

Values: array([4, 7, -5, 3])

Name

Series: Index, Values

2 main Series attribues: Index, Values



```
obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
obj2
```

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

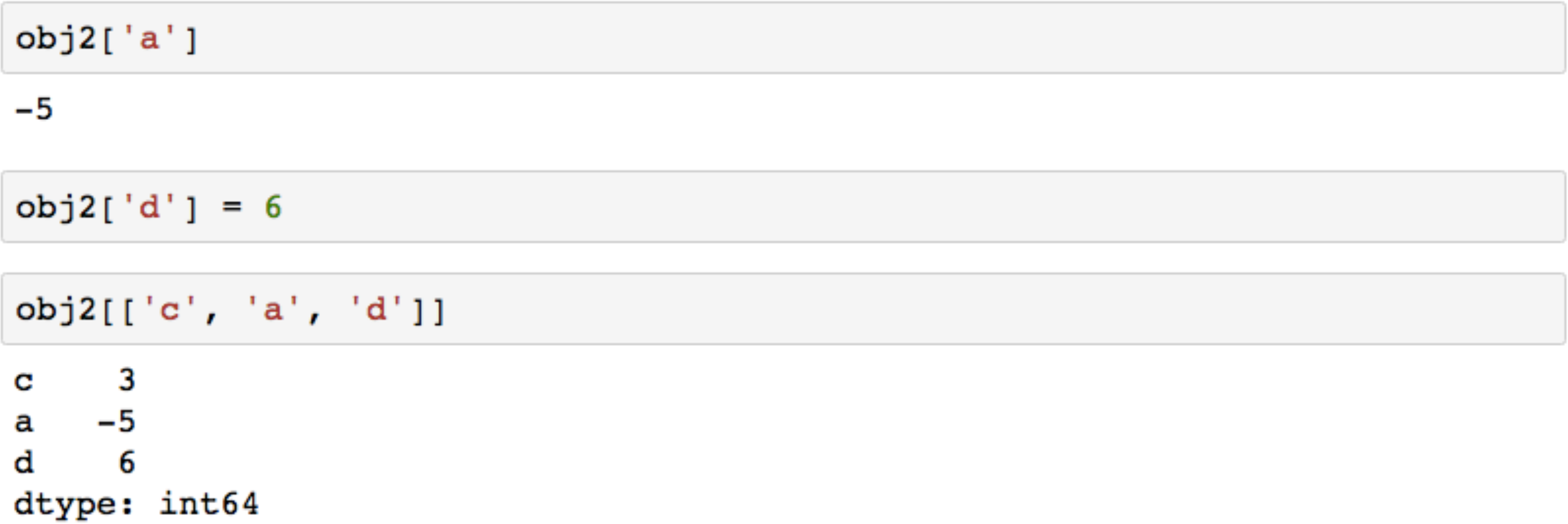
```
obj2.index
```

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

```
obj2.values
```

```
array([ 4,  7, -5,  3])
```

Series: element selection



```
obj2['a']
```

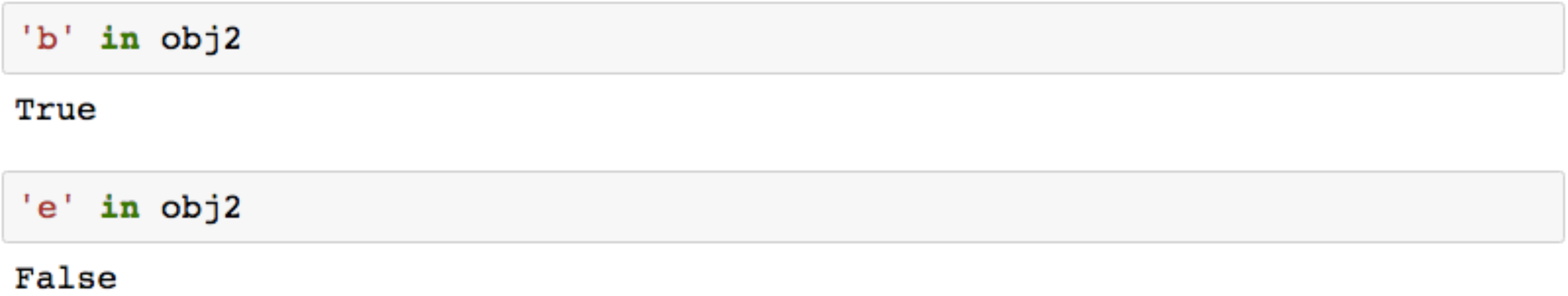
```
-5
```

```
obj2['d'] = 6
```

```
obj2[['c', 'a', 'd']]
```

```
c    3
a   -5
d    6
dtype: int64
```

Series: membership



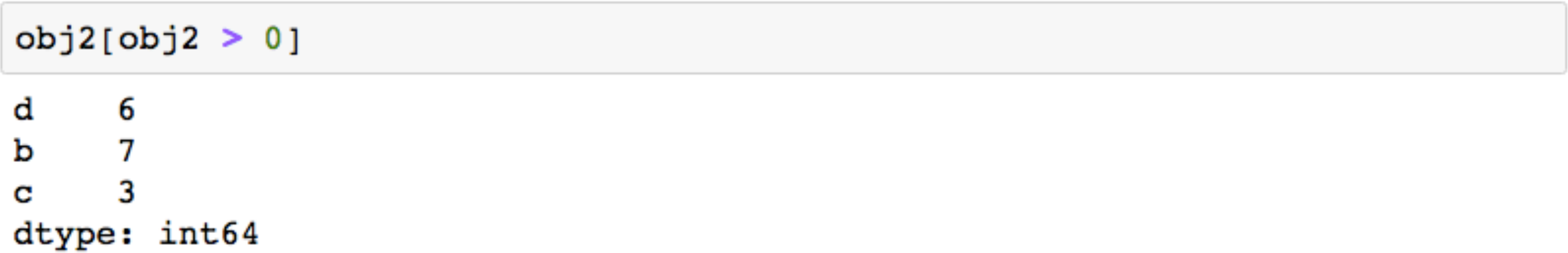
```
'b' in obj2
```

```
True
```

```
'e' in obj2
```

```
False
```

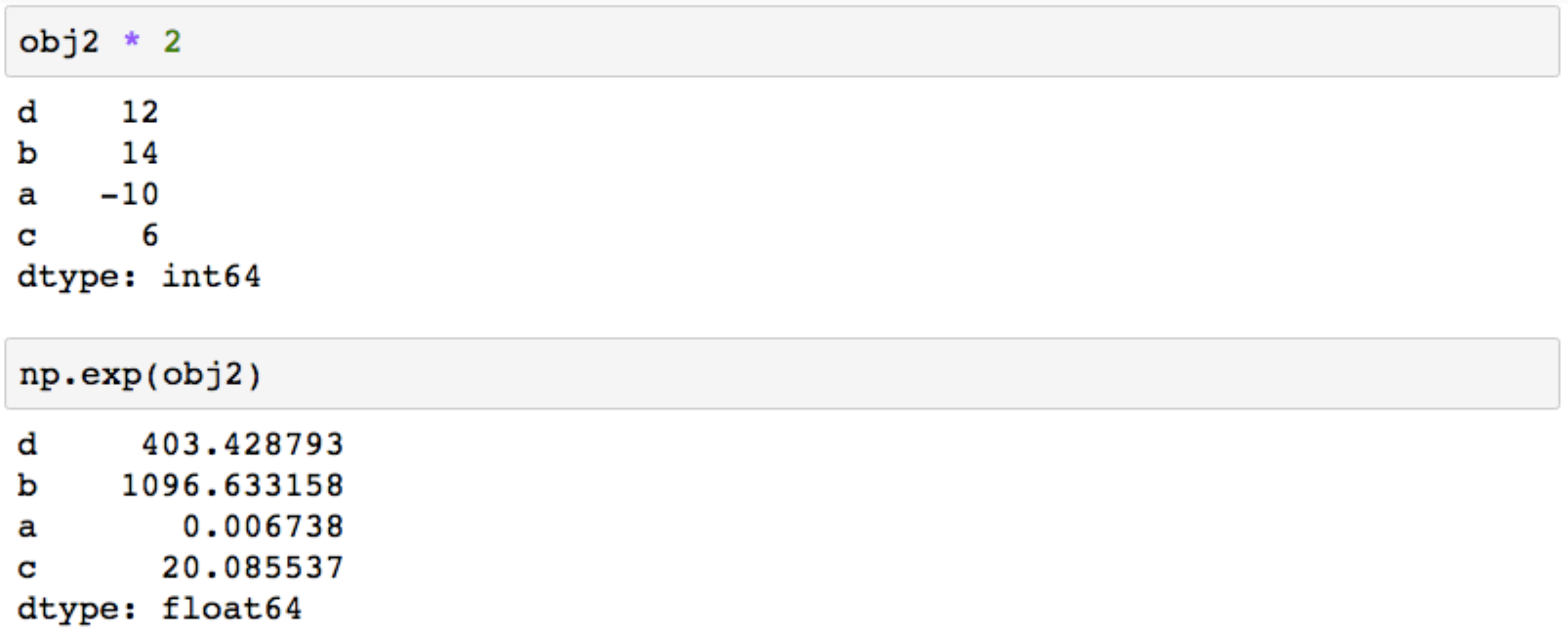
Series: element filtering



```
obj2[obj2 > 0]
```

```
d     6
b     7
c     3
dtype: int64
```

Series: scalar operations



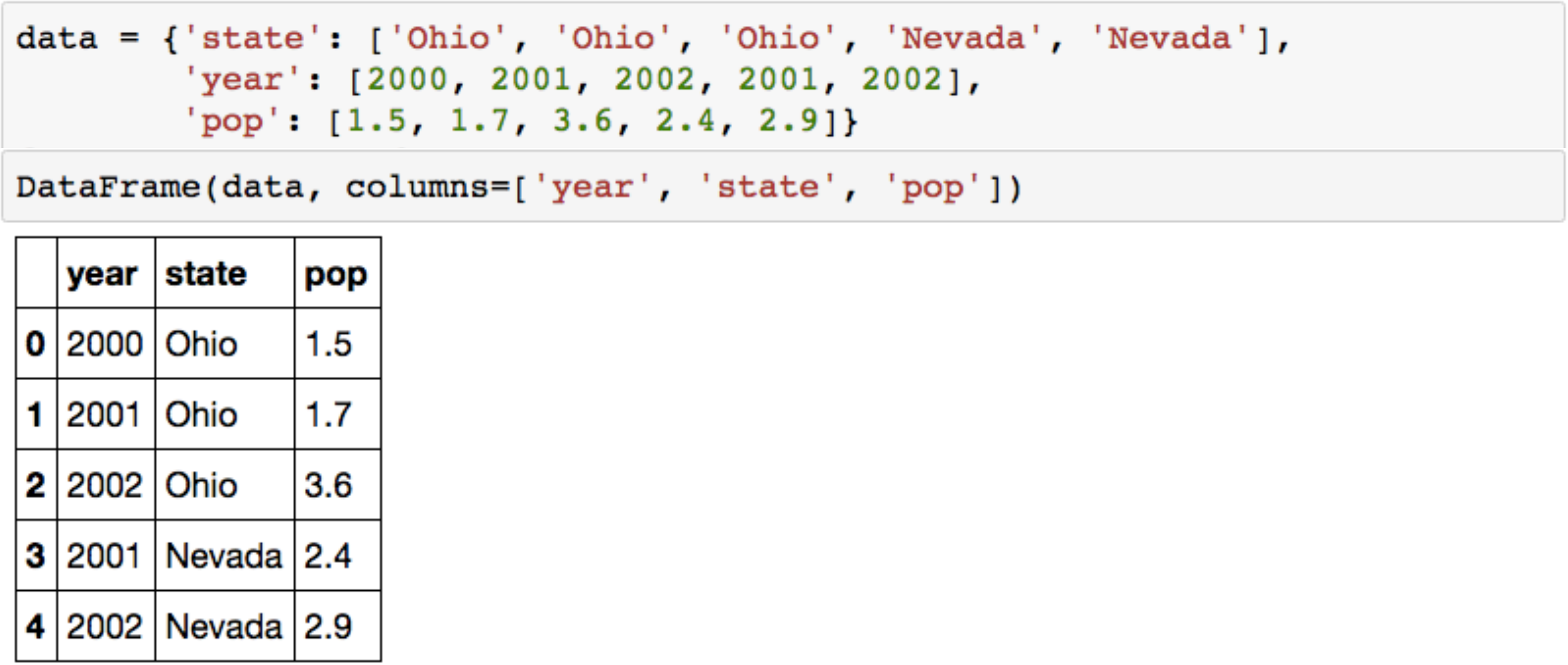
```
obj2 * 2
```

```
d     12
b     14
a    -10
c      6
dtype: int64
```

```
np.exp(obj2)
```

```
d      403.428793
b     1096.633158
a        0.006738
c       20.085537
dtype: float64
```

DataFrame: table in pandas

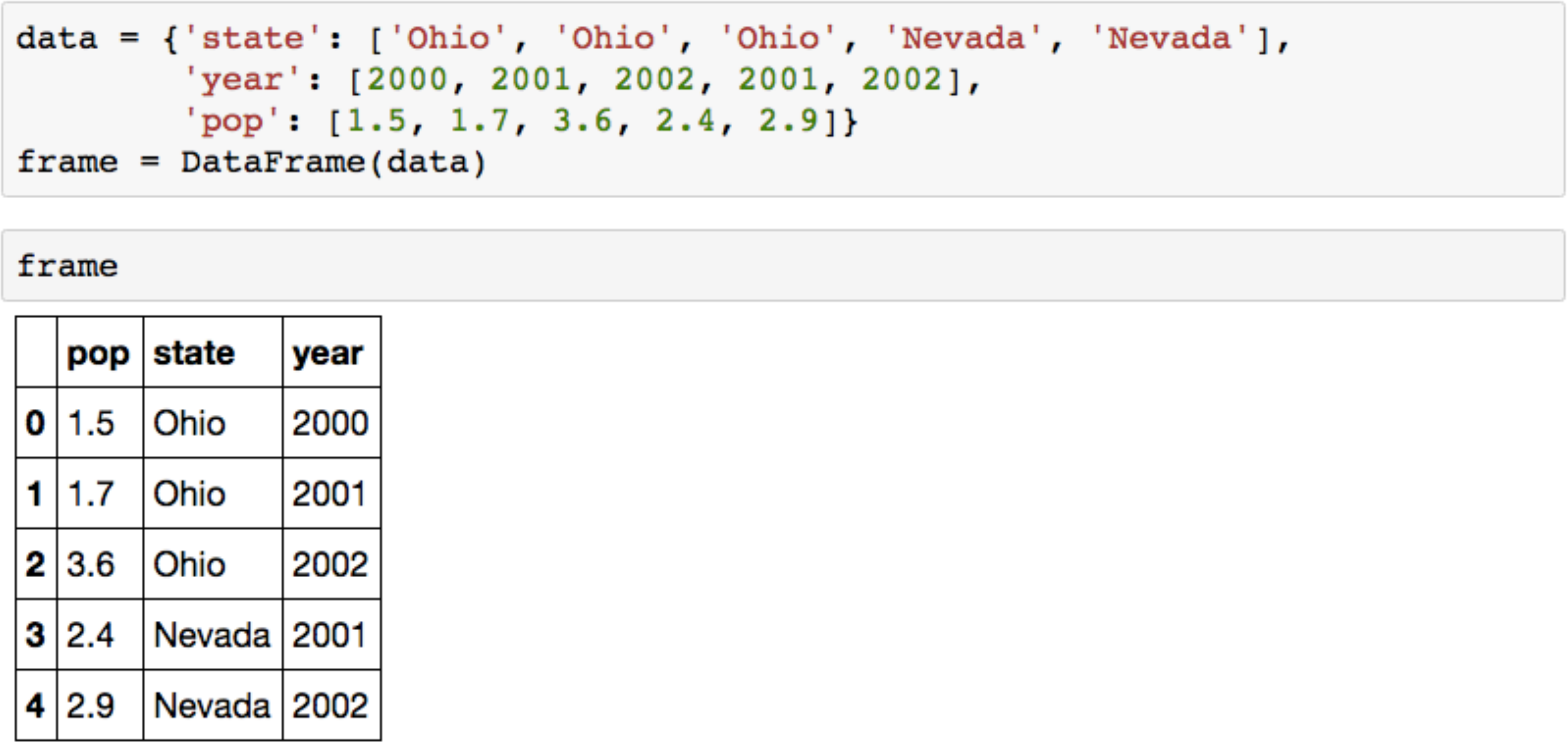


```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
```

```
DataFrame(data, columns=['year', 'state', 'pop'])
```

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

DataFrame: table in pandas

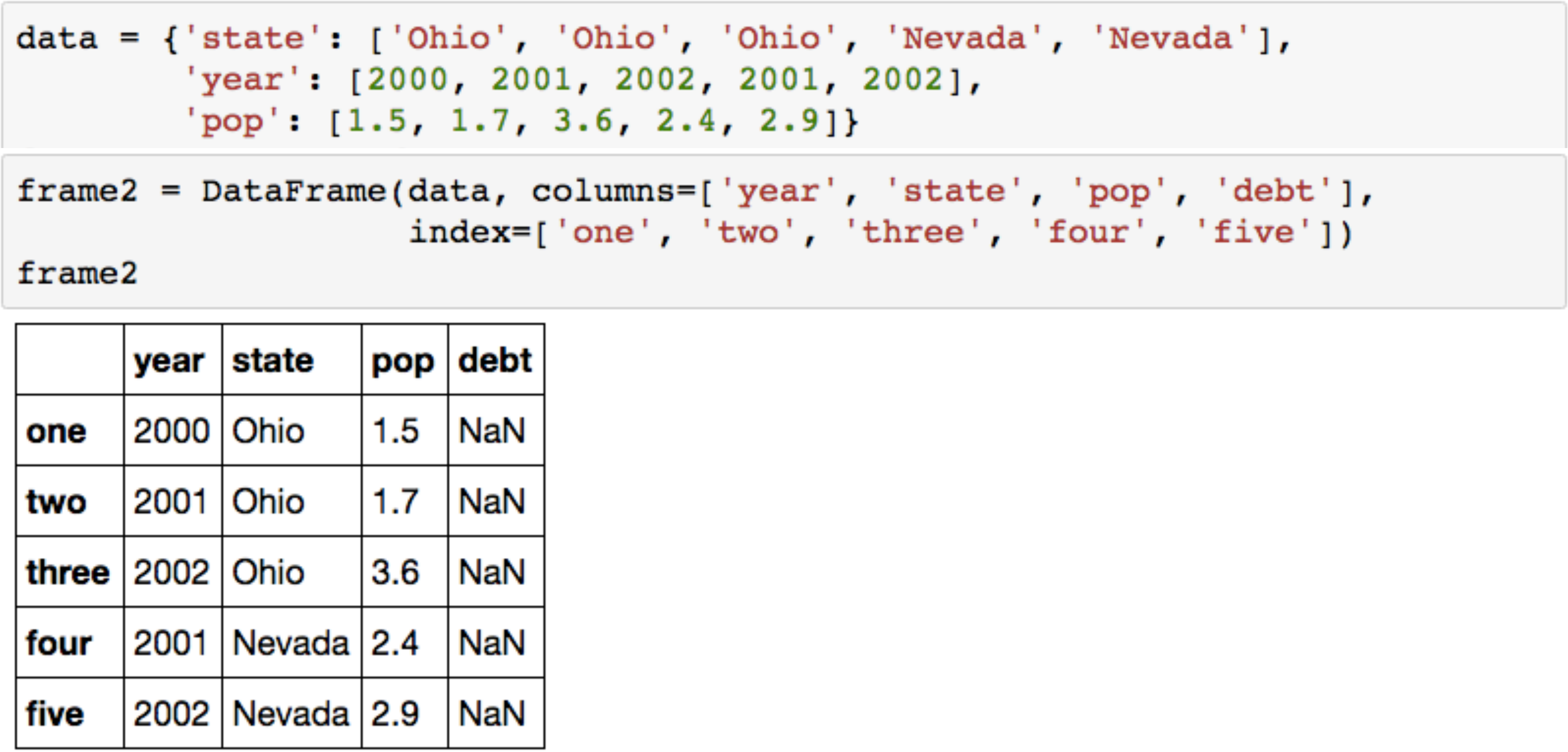


```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
```

```
frame
```

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

DataFrame: columns of lists with indices

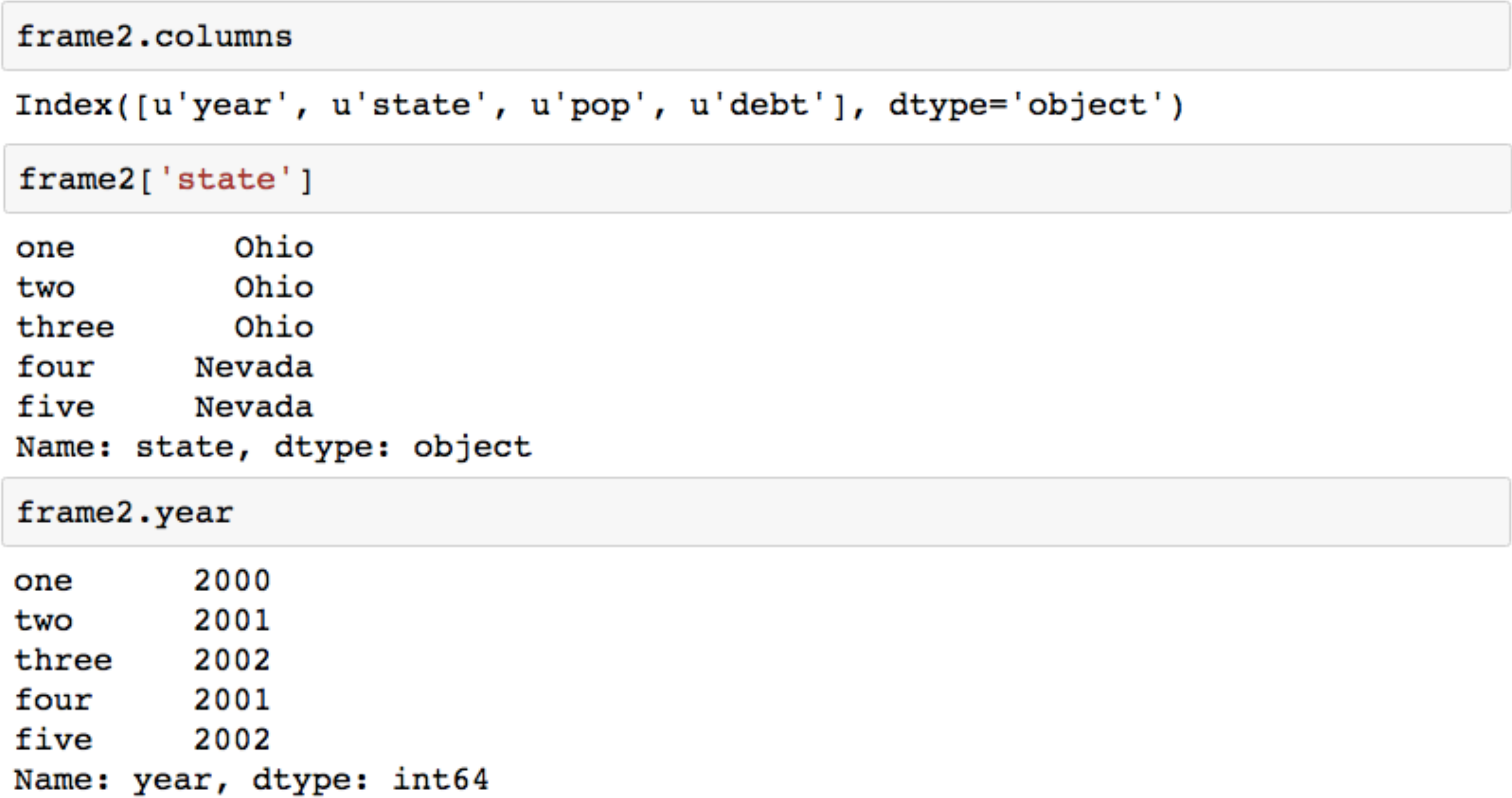


```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
```

```
frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                         index=['one', 'two', 'three', 'four', 'five'])
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN

DataFrame: columns



```
frame2.columns
```

```
Index([u'year', u'state', u'pop', u'debt'], dtype='object')
```

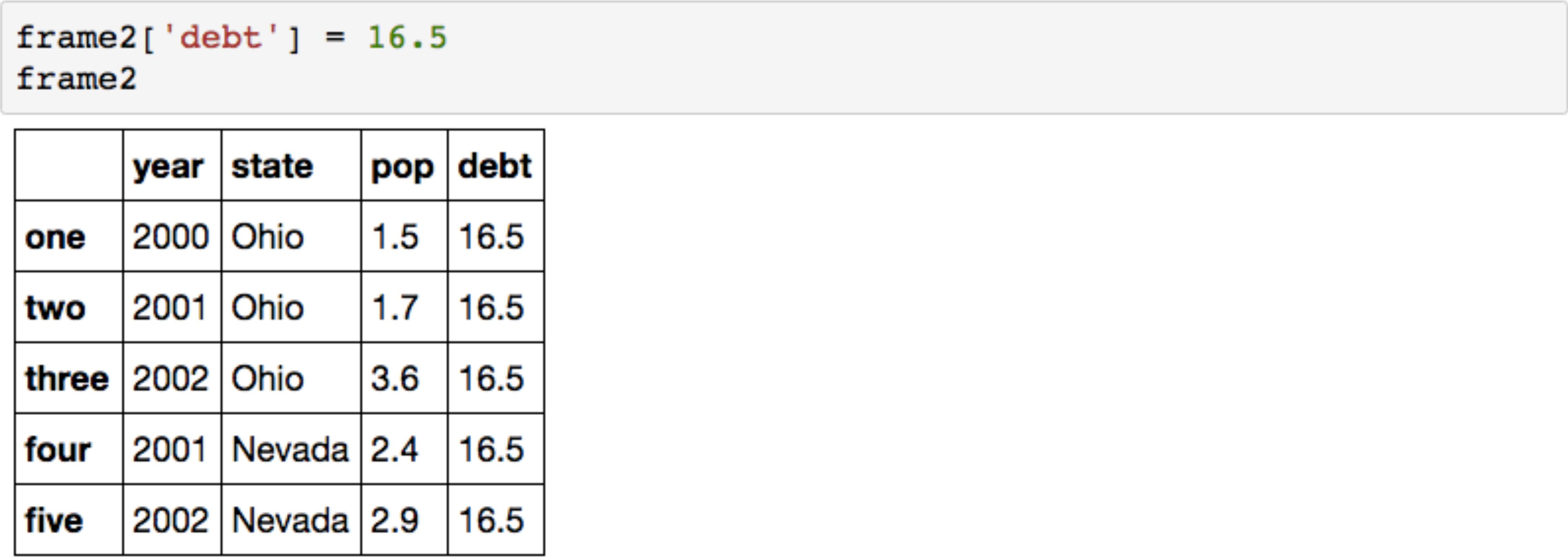
```
frame2['state']
```

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

```
frame2.year
```

```
one      2000
two      2001
three    2002
four     2001
five     2002
Name: year, dtype: int64
```

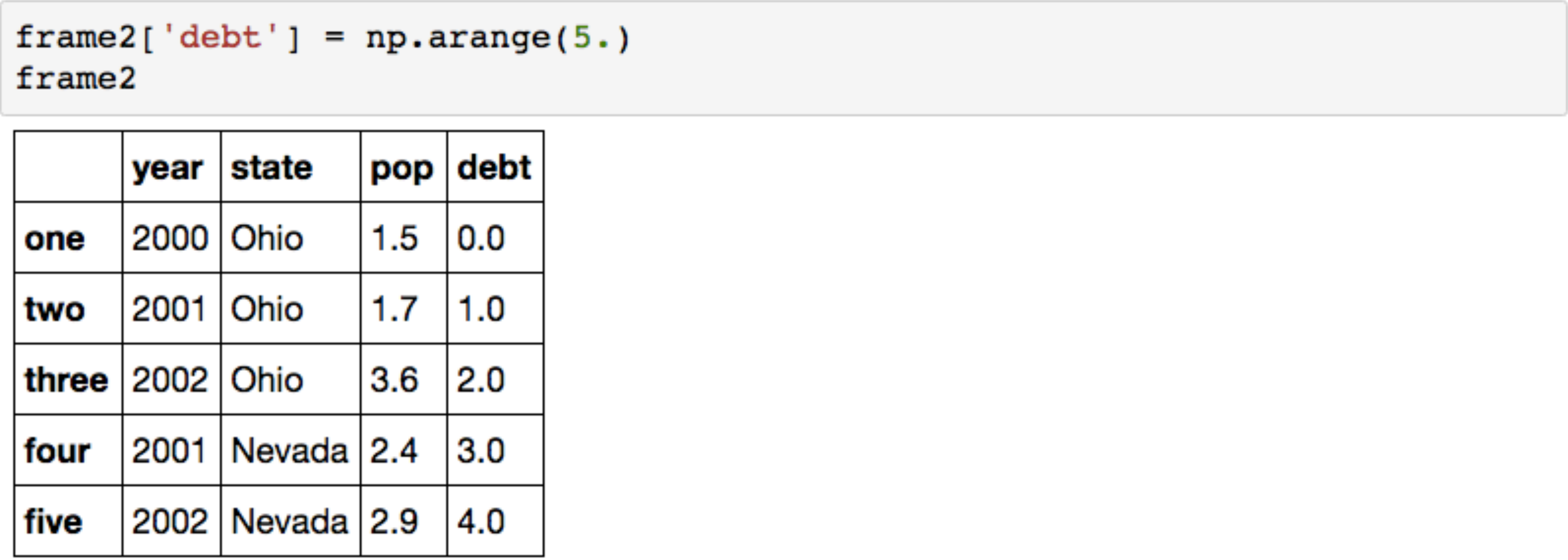
DataFrame: inserting data



```
frame2['debt'] = 16.5
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5

DataFrame: inserting data



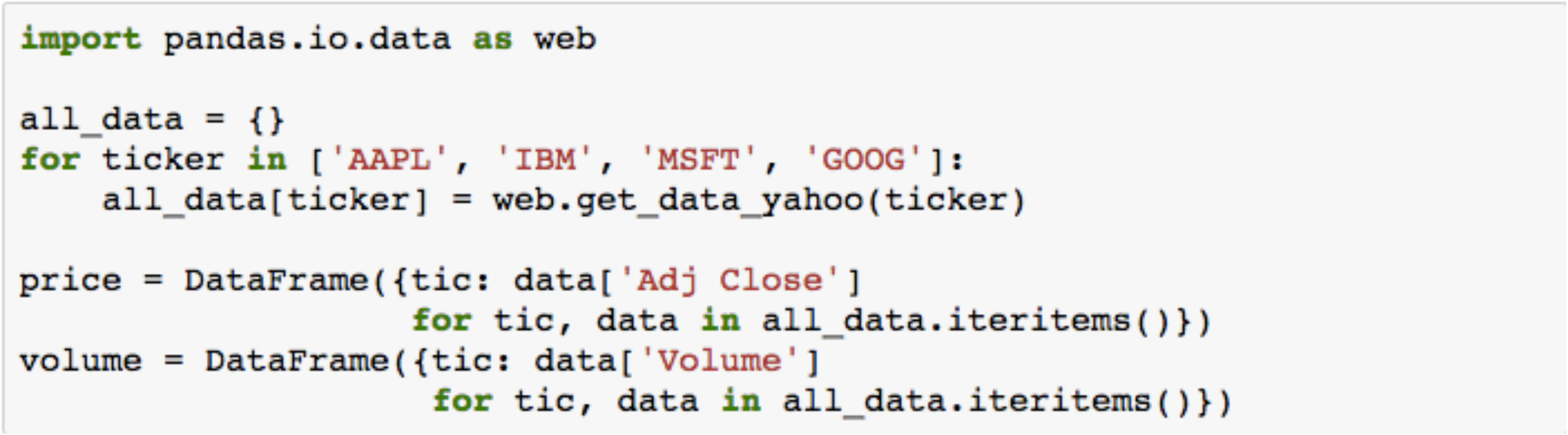
```
frame2['debt'] = np.arange(5.)
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0

Agenda

- Numpy
- Pandas
- **Finance**
- Group Exercise
- Assignment

Getting Yahoo Finance Data

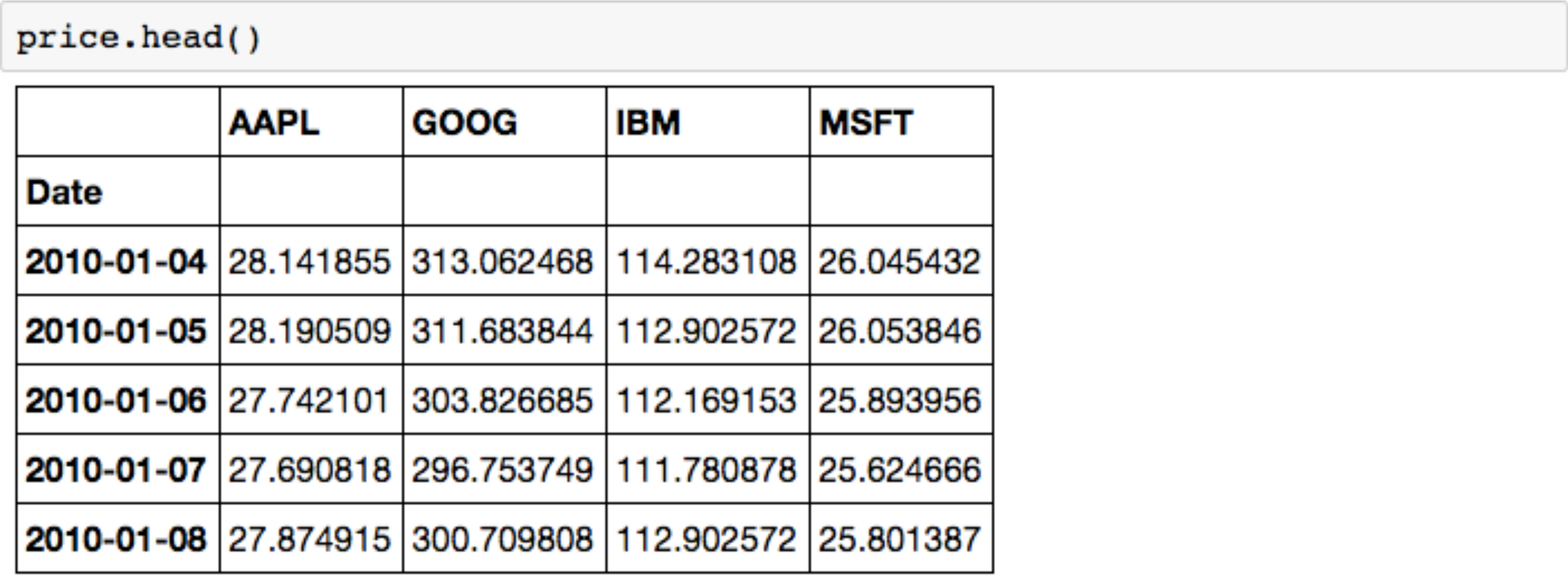


```
import pandas.io.data as web

all_data = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
    all_data[ticker] = web.get_data_yahoo(ticker)

price = DataFrame({tic: data['Adj Close']
                   for tic, data in all_data.iteritems()})
volume = DataFrame({tic: data['Volume']
                    for tic, data in all_data.iteritems()})
```

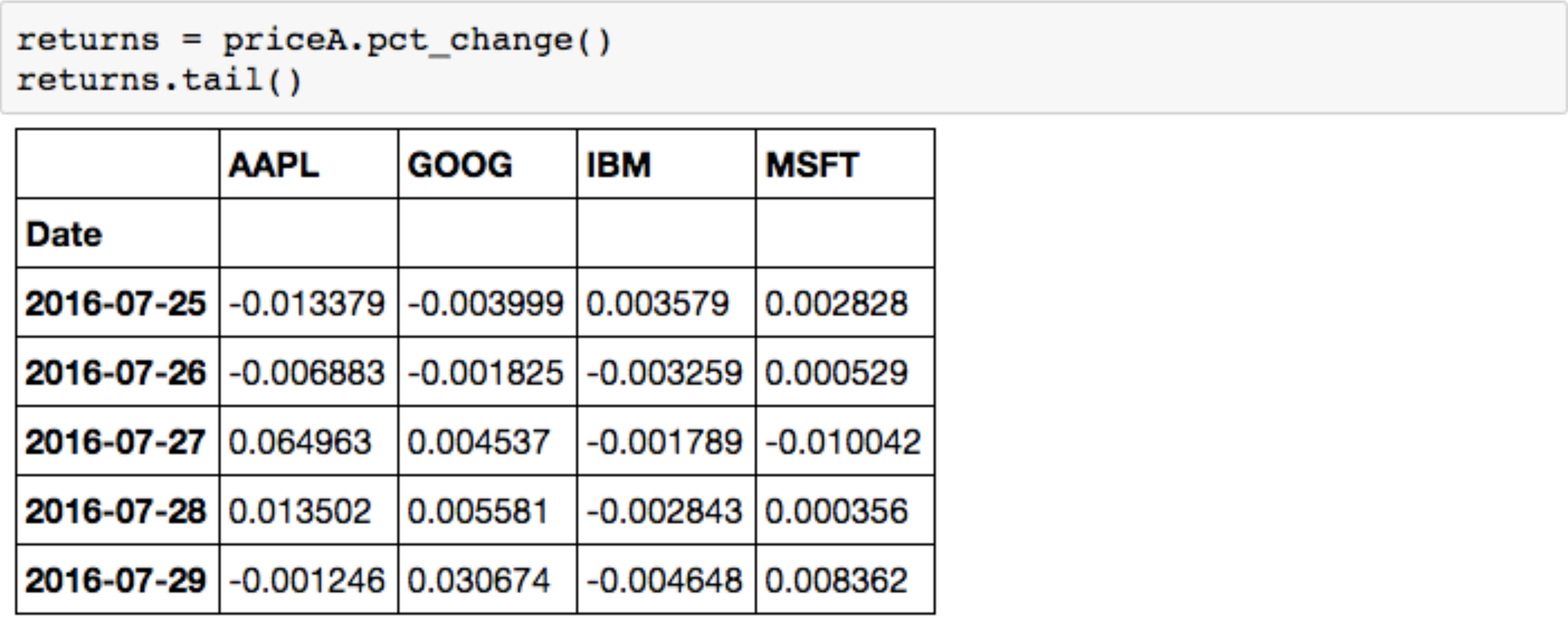
Stock data as DataFrame



```
price.head()
```

	AAPL	GOOG	IBM	MSFT
Date				
2010-01-04	28.141855	313.062468	114.283108	26.045432
2010-01-05	28.190509	311.683844	112.902572	26.053846
2010-01-06	27.742101	303.826685	112.169153	25.893956
2010-01-07	27.690818	296.753749	111.780878	25.624666
2010-01-08	27.874915	300.709808	112.902572	25.801387

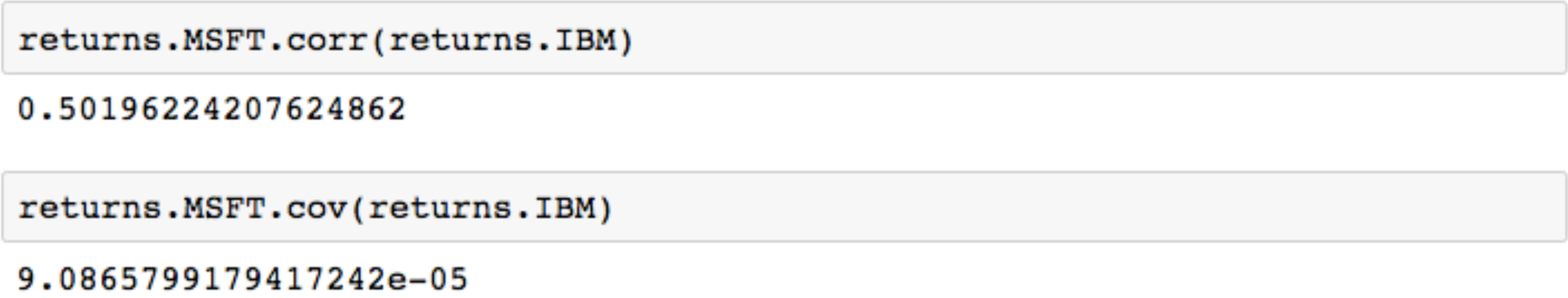
Stock data as DataFrame



```
returns = priceA.pct_change()
returns.tail()
```

	AAPL	GOOG	IBM	MSFT
Date				
2016-07-25	-0.013379	-0.003999	0.003579	0.002828
2016-07-26	-0.006883	-0.001825	-0.003259	0.000529
2016-07-27	0.064963	0.004537	-0.001789	-0.010042
2016-07-28	0.013502	0.005581	-0.002843	0.000356
2016-07-29	-0.001246	0.030674	-0.004648	0.008362

Correlation among stocks



```
returns.MSFT.corr(returns.IBM)
```

```
0.50196224207624862
```

```
returns.MSFT.cov(returns.IBM)
```

```
9.0865799179417242e-05
```

Group Exercise: Dow Jones

- Form groups of 4 students
- **Exercise 1**: find list of all Dow Jones component stock tickers
- **Exercise 2**: search & discuss how the Dow Jones component stock values are related to the Dow Jones Industrial Index value.

Assignment: Dow Jones