CSS 330 Data wrangling and visualization

Lecture 3

Instructor: Nabigazinova Elnura

Data Manipulation with Numpy and pandas

Currently data science is so popular and there are a lot of different tools that can be used for data manipulation.

In this lecture we will focus on the **pandas** library which frequently used to process and manipulate data. This library extensively uses **numpy** library for any mathematical functions so it also should be briefly covered.

Numpy **import numpy as np**

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. (Wikipedia)

Basics

NumPy’s main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In NumPy dimensions are called axes.

NumPy’s array class is called ndarray. It is also known by the alias array. Note that numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality.

Numpy array functionalities:

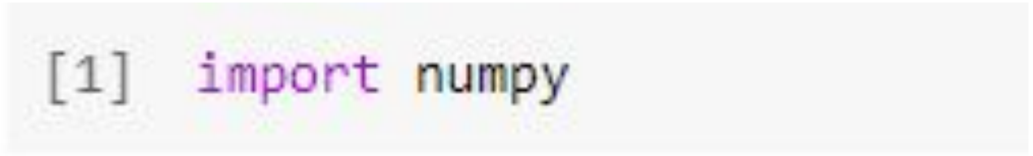
- ndarray, a fast and space-efficient multidimensional array providing vectorized arithmetic operations and sophisticated broadcasting capabilities
- Standard mathematical functions for fast operations on entire arrays of data without having to write loops
- Tools for reading / writing array data to disk and working with memory-mapped files
- Linear algebra, random number generation, and Fourier transform capabilities
- Tools for integrating code written in C, C++, and Fortran

Numpy array functionalities:

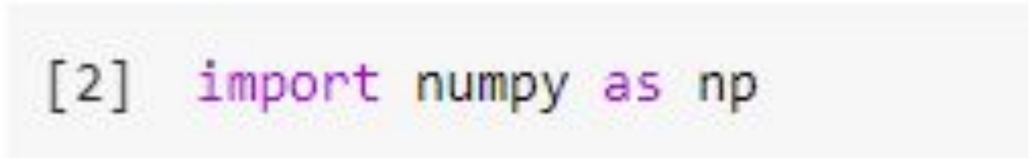
The more important attributes of an ndarray object are:

- **ndarray.ndim** - the number of axes (dimensions) of the array.
- **ndarray.shape** - the dimensions of the array. For a matrix with *n* rows and *m* columns, shape will be (n,m).
- **ndarray.size** - the total number of elements of the array. This is equal to the product of the elements of shape.
- **ndarray.dtype** - an object describing the type of the elements in the array.
- **ndarray.itemsize** - the size in bytes of each element of the array.
- **ndarray.data** - the buffer containing the actual elements of the array. Normally, we won’t need to use this attribute because we will access the elements in an array using indexing facilities.

Importing numpy



```
[1]  import numpy
```

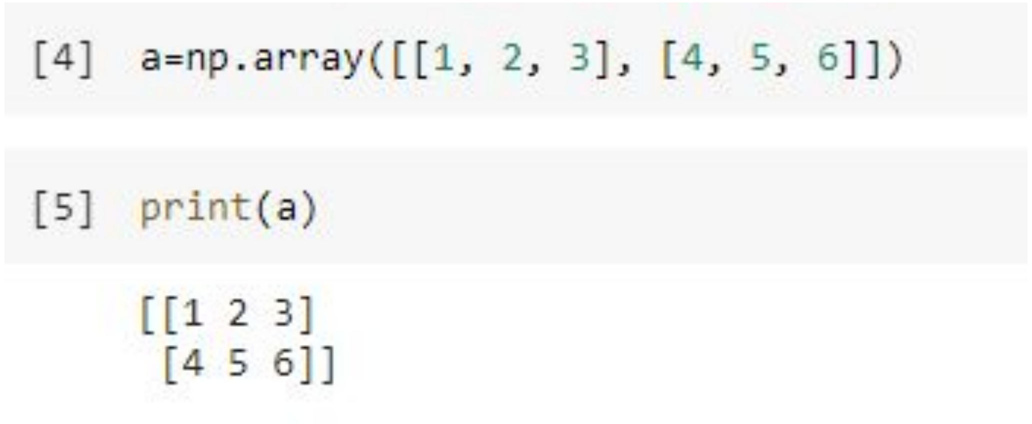


```
[2]  import numpy as np
```

The NumPy ndarray: A Multidimensional Array Object

One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large data sets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

NumPy ndarray:

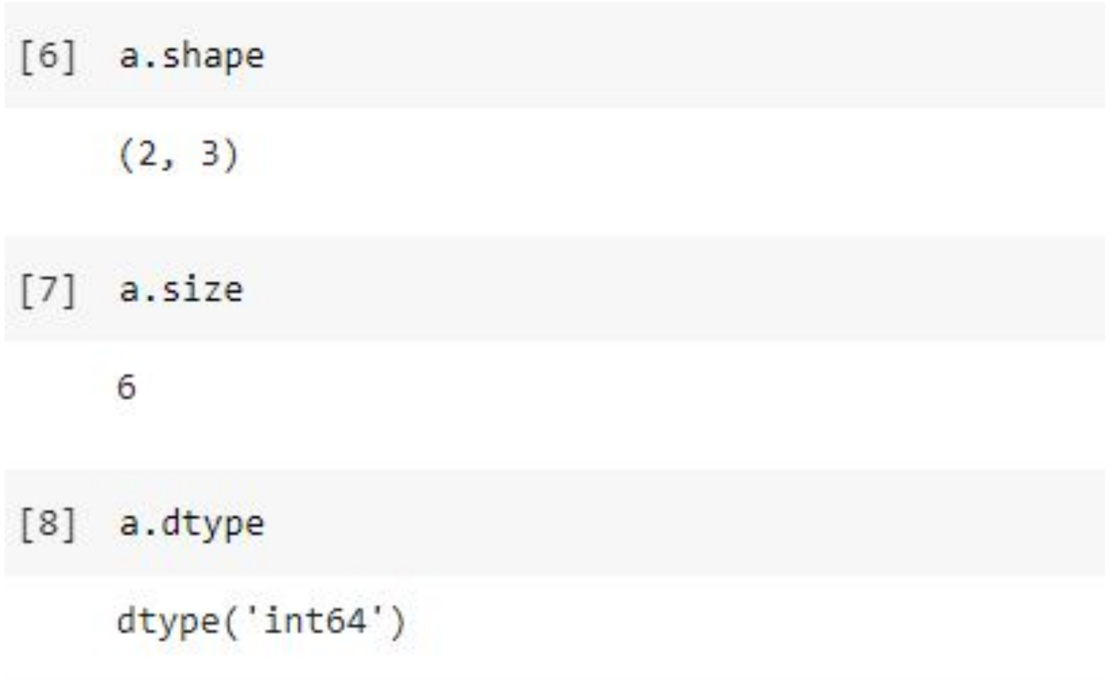


```
[4]  a=np.array([[1, 2, 3], [4, 5, 6]])
```

```
[5]  print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

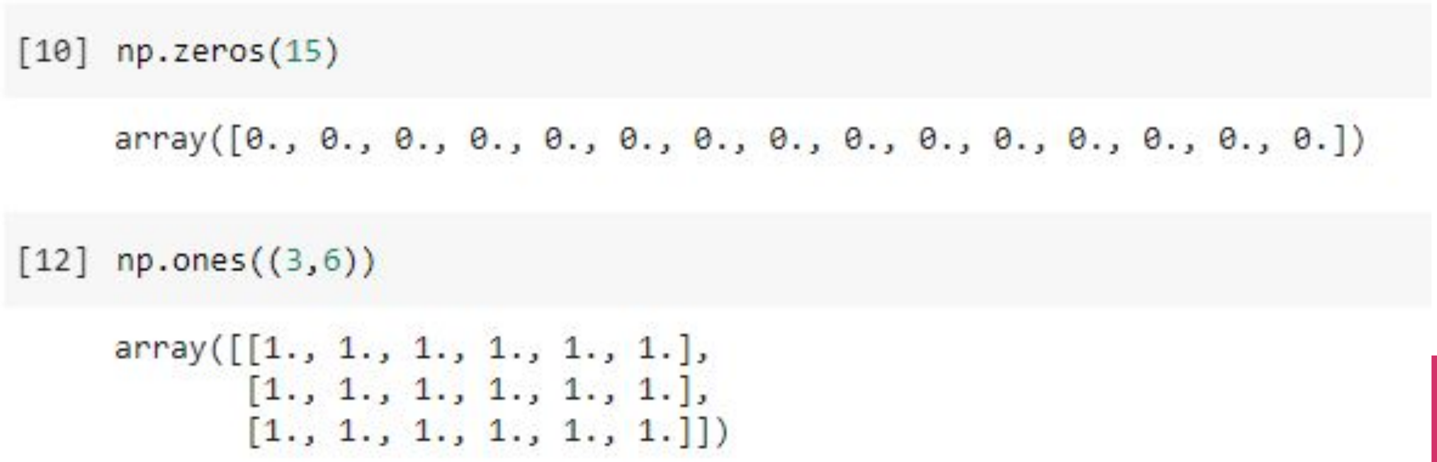
NumPy ndarray:



```
[6]  a.shape

     (2, 3)

[7]  a.size

     6

[8]  a.dtype

     dtype('int64')
```

Numpy zeros and ones

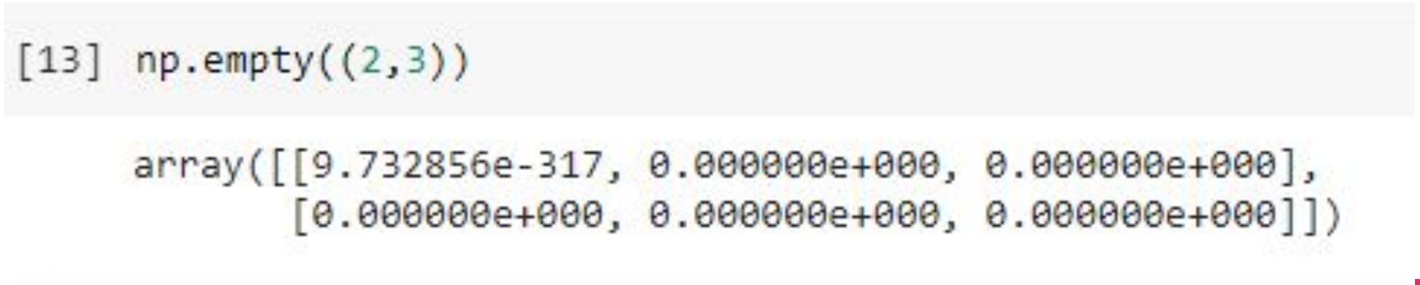
In addition to np.array , there are a number of other functions for creating new arrays. As examples, zeros and ones create arrays of 0's or 1's, respectively, with a given length or shape.



```
[10] np.zeros(15)

     array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

[12] np.ones((3,6))

     array([[1., 1., 1., 1., 1., 1.],
            [1., 1., 1., 1., 1., 1.],
            [1., 1., 1., 1., 1., 1.]])
```

Numpy empty

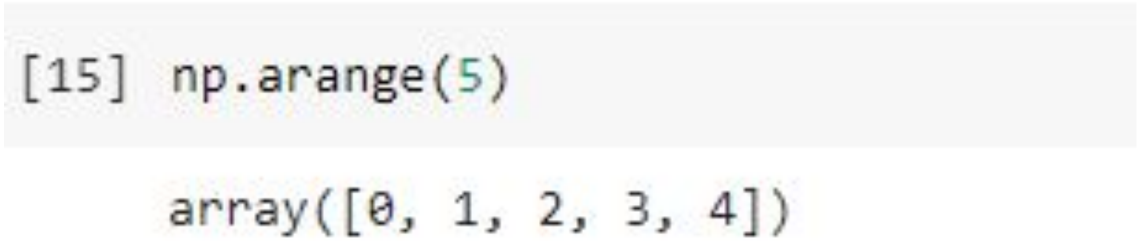
Numpy empty creates an array without initializing its values to any particular value. *It's not safe to assume that np.empty will return an array of all zeros. In many cases, as shown below, it will return uninitialized garbage values.*



```
[13] np.empty((2,3))

     array([[9.732856e-317, 0.000000e+000, 0.000000e+000],
            [0.000000e+000, 0.000000e+000, 0.000000e+000]])
```

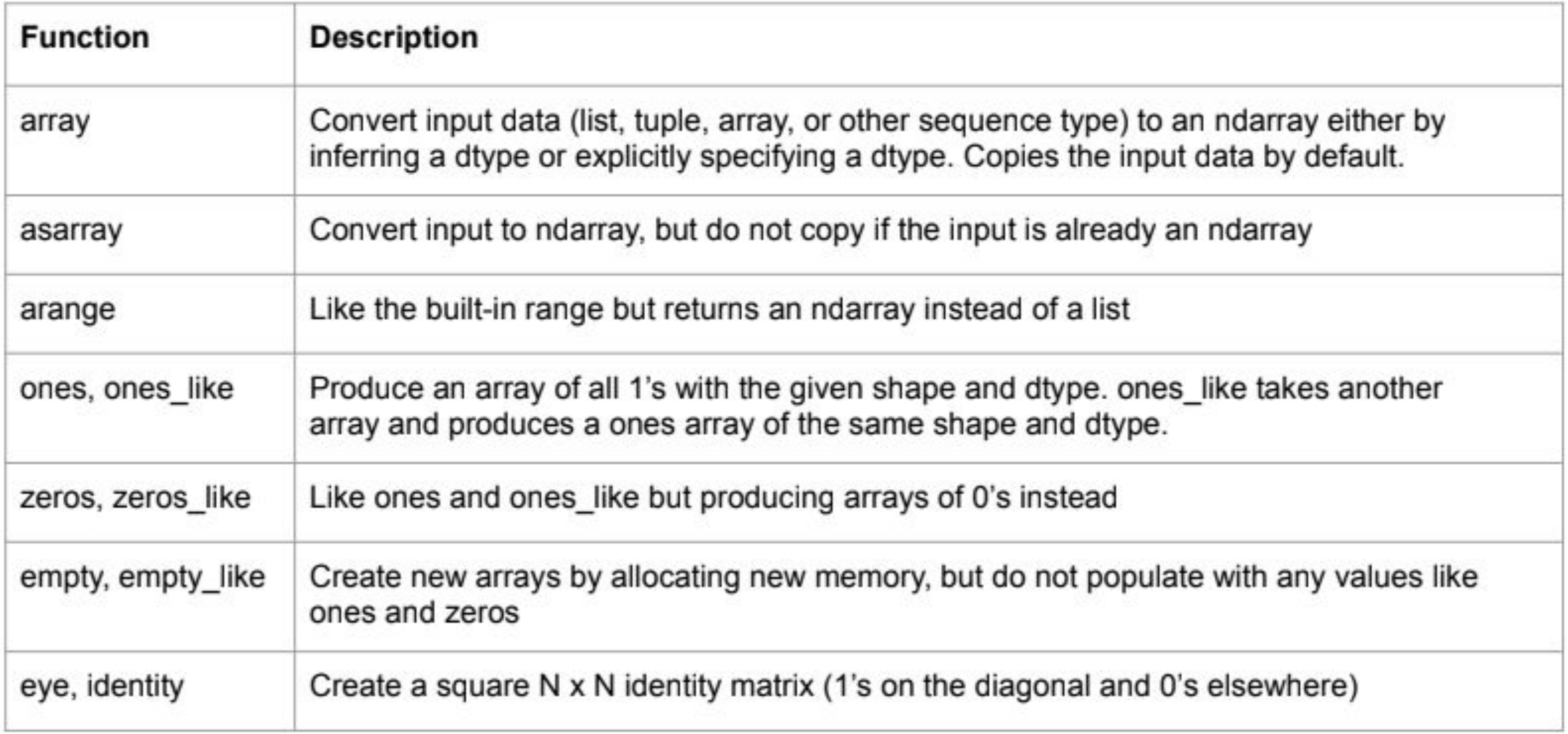
Numpy arange

Numpy arange is an array-valued version of the built-in Python range function.



```
[15] np.arange(5)

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

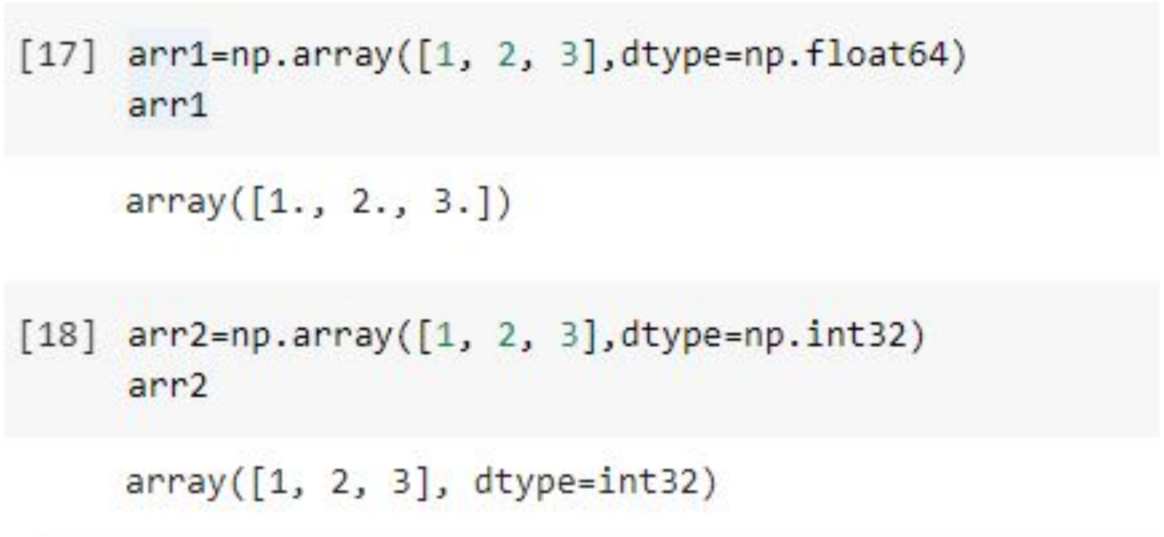
Array creation functions



Function	Description
array	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.
asarray	Convert input to ndarray, but do not copy if the input is already an ndarray
arange	Like the built-in range but returns an ndarray instead of a list
ones, ones_like	Produce an array of all 1's with the given shape and dtype. ones_like takes another array and produces a ones array of the same shape and dtype.
zeros, zeros_like	Like ones and ones_like but producing arrays of 0's instead
empty, empty_like	Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
eye, identity	Create a square N x N identity matrix (1's on the diagonal and 0's elsewhere)

Data Types for ndarrays

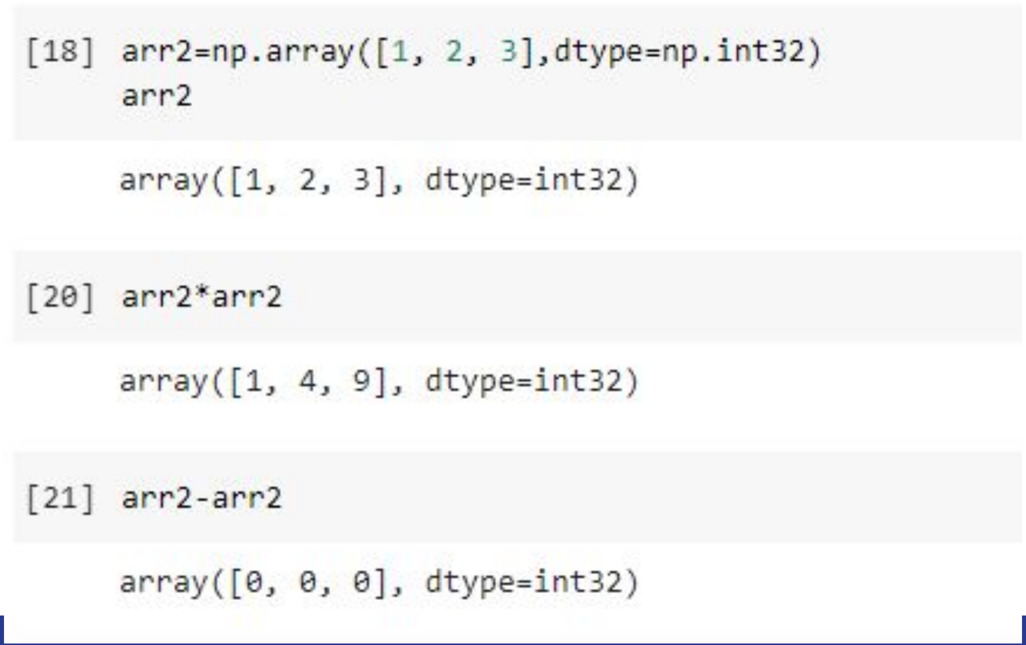
The data type or dtype is a special object containing the information the ndarray needs to interpret a chunk of memory as a particular type of data.



```
[17] arr1=np.array([1, 2, 3],dtype=np.float64)
     arr1

     array([1., 2., 3.])

[18] arr2=np.array([1, 2, 3],dtype=np.int32)
     arr2

     array([1, 2, 3], dtype=int32)
```

Operations between Arrays and Scalars

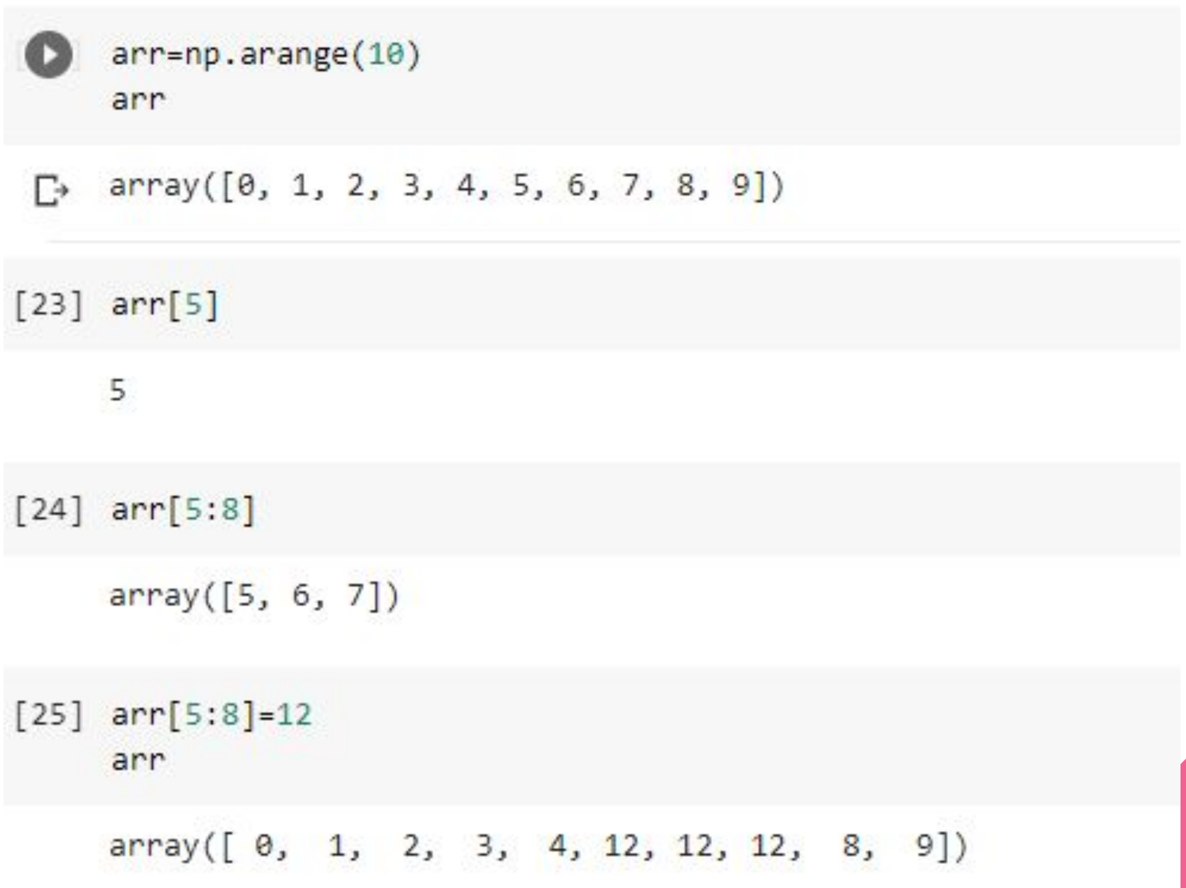
Arrays are important because they enable you to express batch operations on data without writing any for loops. This is usually called vectorization. Any arithmetic operations between equal-size arrays applies the operation elementwise.



```
[18] arr2=np.array([1, 2, 3],dtype=np.int32)
     arr2

     array([1, 2, 3], dtype=int32)

[20] arr2*arr2

     array([1, 4, 9], dtype=int32)

[21] arr2-arr2

     array([0, 0, 0], dtype=int32)
```

Basic Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists.



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

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

```
[23] arr[5]
```

```
5
```

```
[24] arr[5:8]
```

```
array([5, 6, 7])
```

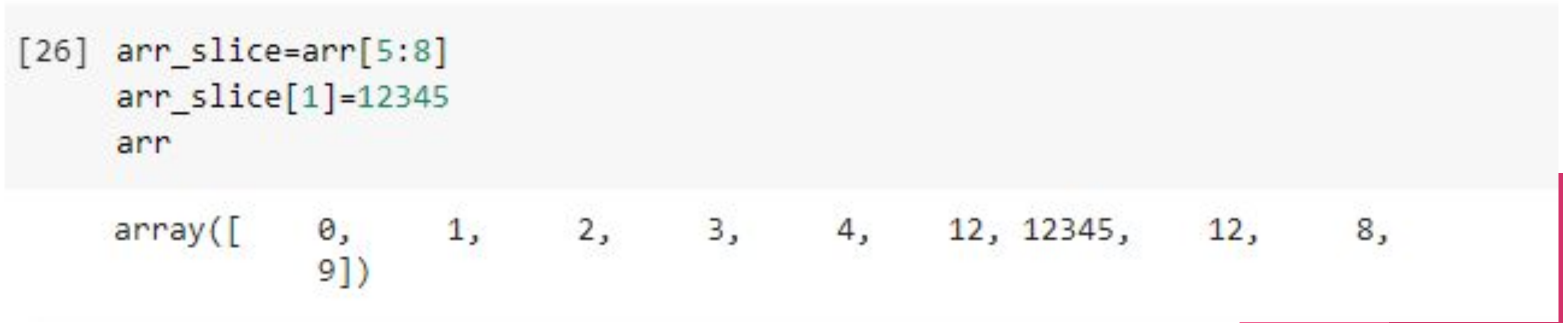
```
[25] arr[5:8]=12
     arr
```

```
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

Basic indexing and Slicing

An important first distinction from lists is that array slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array

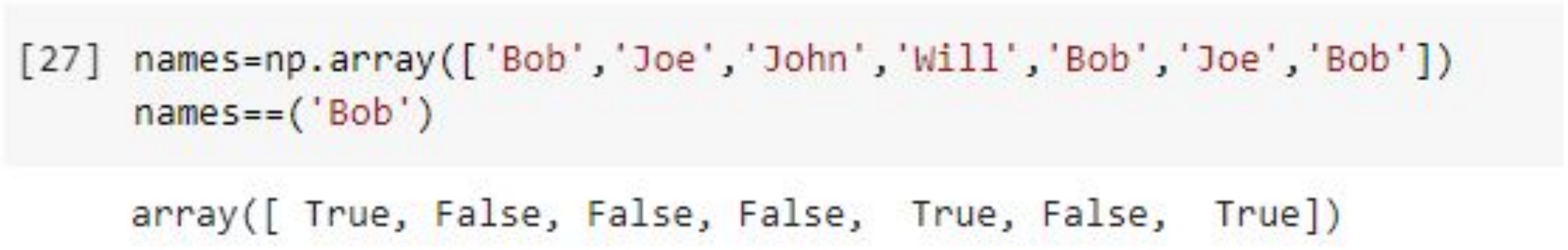
If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array; for example arr[5:8].copy()



```
[26] arr_slice=arr[5:8]
     arr_slice[1]=12345
     arr

     array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
                9])
```

Boolean Indexing

Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array



```
[27] names=np.array(['Bob','Joe','John','Will','Bob','Joe','Bob'])
     names==('Bob')

     array([ True, False, False, False,  True, False,  True])
```

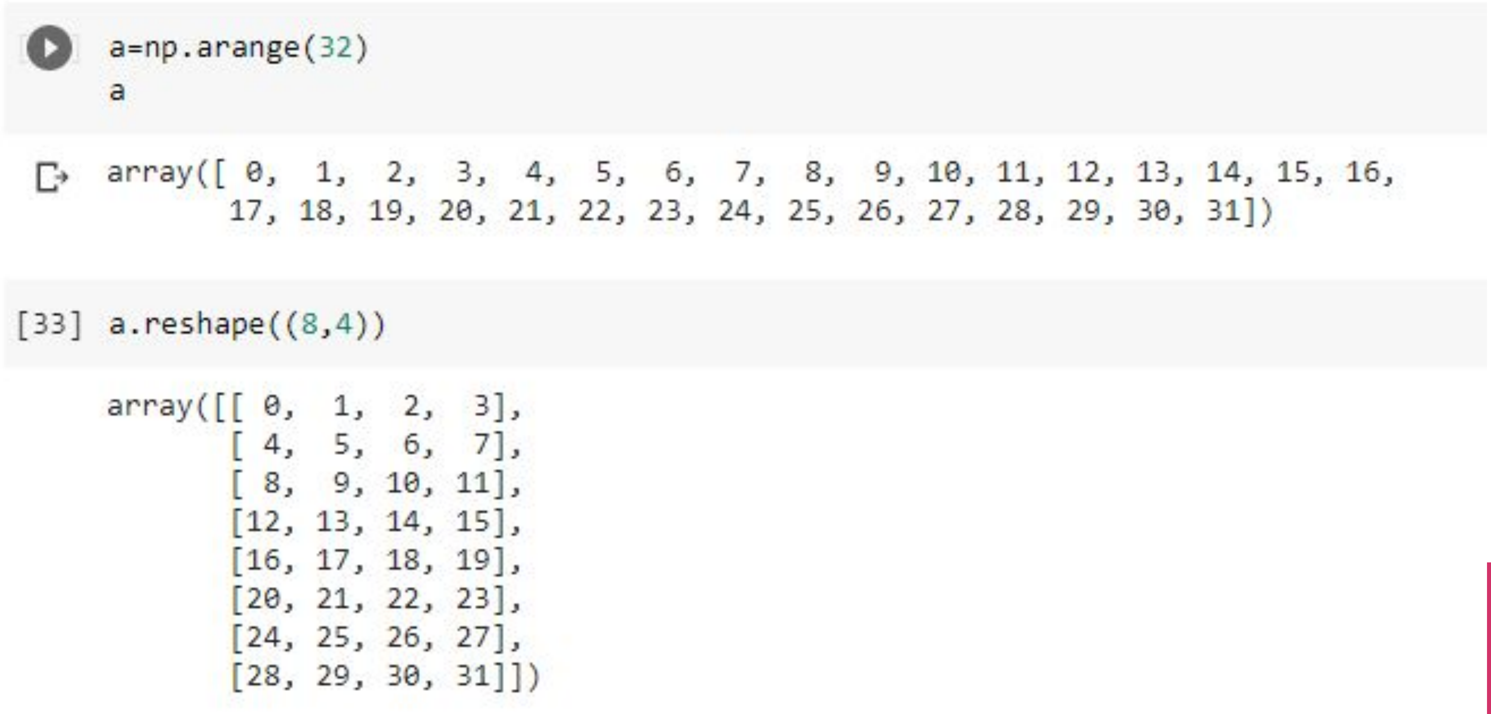
Fancy indexing and index tricks

NumPy offers more indexing facilities than regular Python sequences.

In addition to indexing by integers and slices, as we saw before, arrays can be indexed by arrays of integers and arrays of booleans.

```
>>> a = np.arange(12)**2                    # the first 12 square numbers
>>> i = np.array( [ 1,1,3,8,5 ] )           # an array of indices
>>> a[i]                                    # the elements of a at the positions i
array([ 1,  1,  9, 64, 25])
```

reshape



```
a=np.arange(32)
a
```

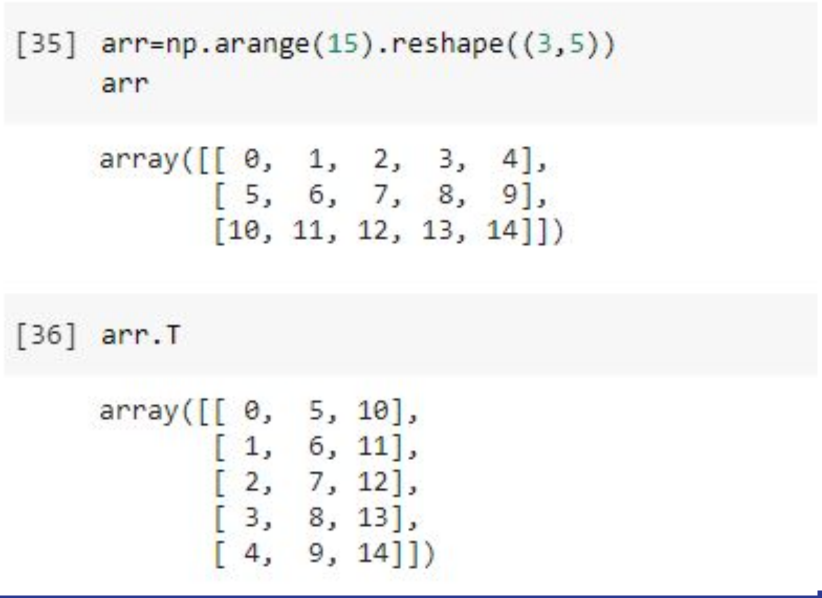
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31])
```

```
[33] a.reshape((8,4))
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

Transposing Arrays

Transposing is a special form of reshaping which similarly returns a view on the underlying data without copying anything. Arrays have the transpose method and also the special T attribute



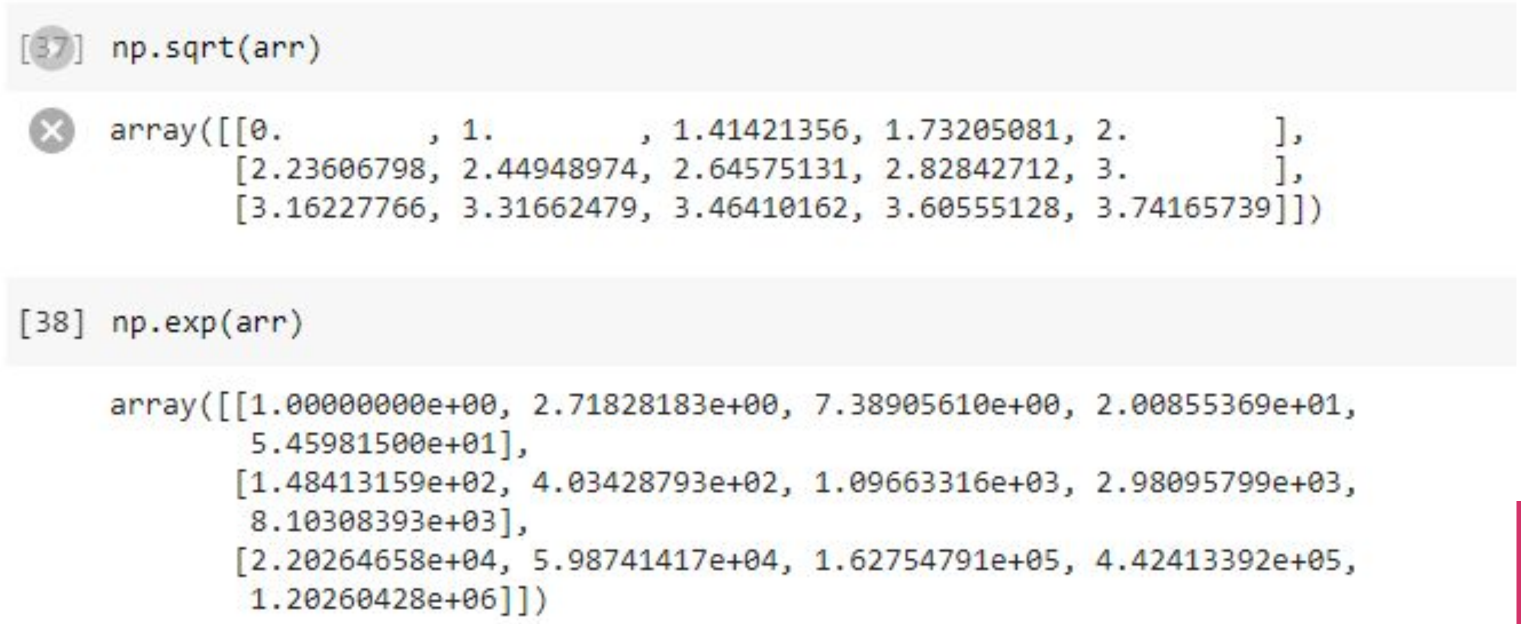
```
[35] arr=np.arange(15).reshape((3,5))
     arr

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

[36] arr.T

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

Universal Functions:
Fast Element-wise Array Functions

A universal function, or ufunc, is a function that performs element wise operations on data in ndarrays.

You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

Universal Functions:
Fast Element-wise Array Functions



```
[37] np.sqrt(arr)
```

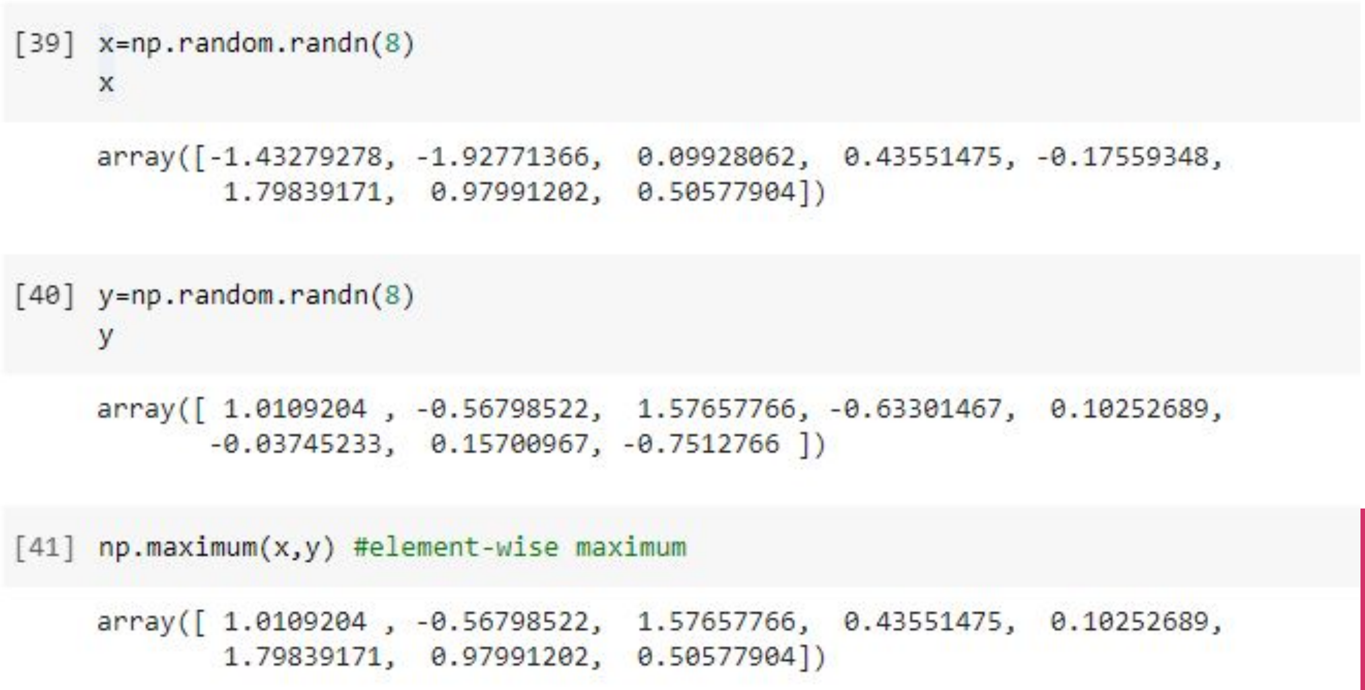
```
array([[0.        , 1.        , 1.41421356, 1.73205081, 2.        ],
       [2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ],
       [3.16227766, 3.31662479, 3.46410162, 3.60555128, 3.74165739]])
```

```
[38] np.exp(arr)
```

```
array([[1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
        5.45981500e+01],
       [1.48413159e+02, 4.03428793e+02, 1.09663316e+03, 2.98095799e+03,
        8.10308393e+03],
       [2.20264658e+04, 5.98741417e+04, 1.62754791e+05, 4.42413392e+05,
        1.20260428e+06]])
```

Binary ufuncs

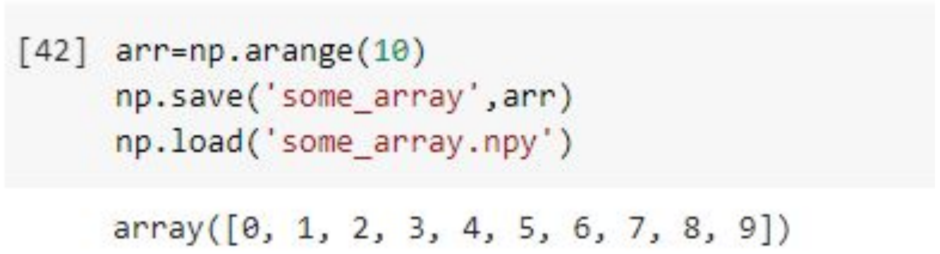
Binary ufuncs take 2 arrays and return a single array as the result



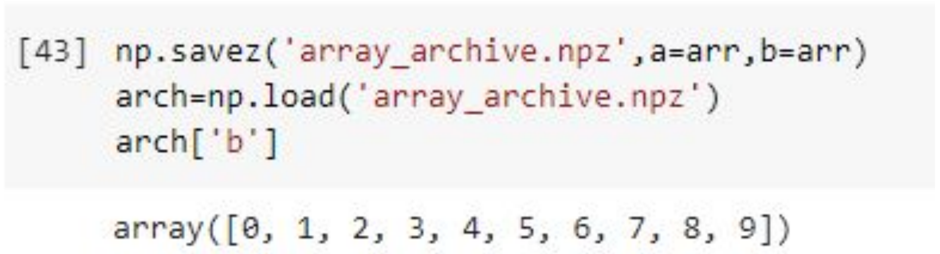
```
[39] x=np.random.randn(8)
     x

     array([-1.43279278, -1.92771366,  0.09928062,  0.43551475, -0.17559348,
             1.79839171,  0.97991202,  0.50577904])

[40] y=np.random.randn(8)
     y

     array([ 1.0109204 , -0.56798522,  1.57657766, -0.63301467,  0.10252689,
            -0.03745233,  0.15700967, -0.7512766 ])

[41] np.maximum(x,y) #element-wise maximum

     array([ 1.0109204 , -0.56798522,  1.57657766,  0.43551475,  0.10252689,
             1.79839171,  0.97991202,  0.50577904])
```

Storing Arrays on Disk in Binary Format

np.save and np.load are the two workhorse functions for efficiently saving and loading array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension .npy



```
[42] arr=np.arange(10)
     np.save('some_array',arr)
     np.load('some_array.npy')

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



```
[43] np.savez('array_archive.npz',a=arr,b=arr)
     arch=np.load('array_archive.npz')
     arch['b']

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

Pandas **from pandas import pd**

In computer programming, pandas is a software library written for the Python programming language for data manipulation and analysis.

In particular, it offers data structures and operations for manipulating numerical tables and time series. It is free software released under the three-clause BSD license.

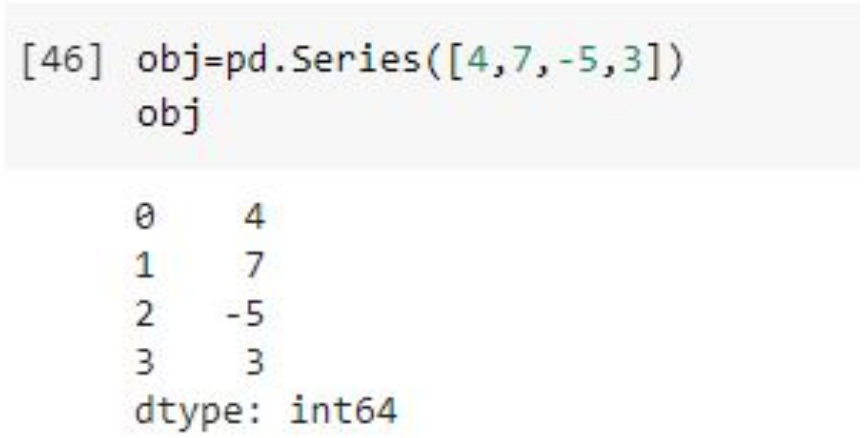
The name is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals.
(Wikipedia)

Pandas elements

- A set of labeled array data structures, the primary of which are Series and DataFrame.
- Index objects enabling both simple axis indexing and multi-level / hierarchical axis indexing.
- An integrated group by engine for aggregating and transforming data sets.
- Date range generation (date_range) and custom date offsets enabling the implementation of customized frequencies.
- Input/Output tools: loading tabular data from flat files (CSV, delimited, Excel 2003), and saving and loading pandas objects from the fast and efficient PyTables/HDF5 format.
- Memory-efficient "sparse" versions of the standard data structures for storing data that is mostly missing or mostly constant (some fixed value).
- Moving window statistics (rolling mean, rolling standard deviation, etc.).

Series

A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its index. The simplest Series is formed from only an array of data

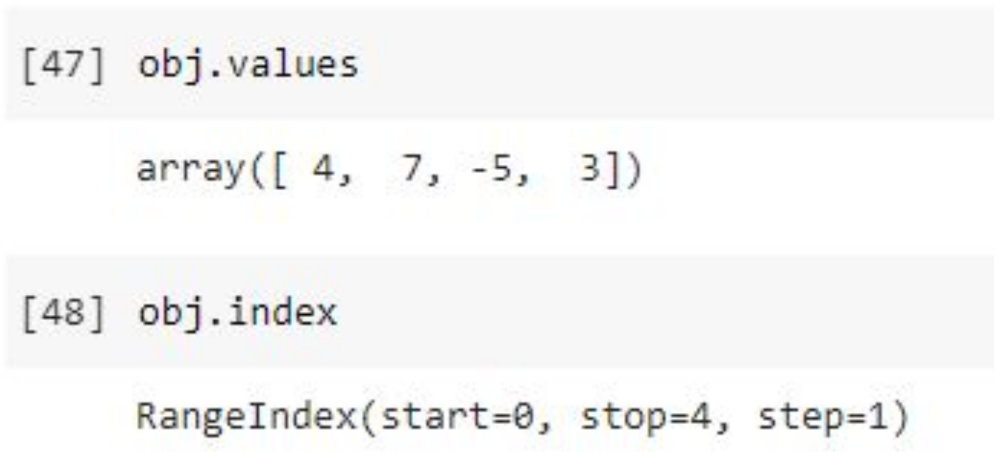


```
[46] obj=pd.Series([4,7,-5,3])
     obj
```

```
0    4
1    7
2   -5
3    3
dtype: int64
```

Series

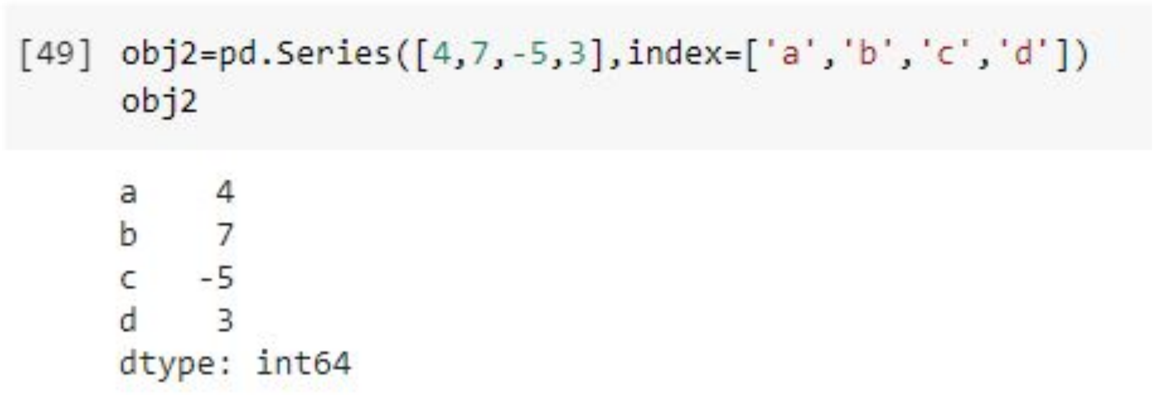
Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data)



```
[47] obj.values

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

[48] obj.index

     RangeIndex(start=0, stop=4, step=1)
```

Series indexing

Often it will be desirable to create a Series with an index identifying each data point

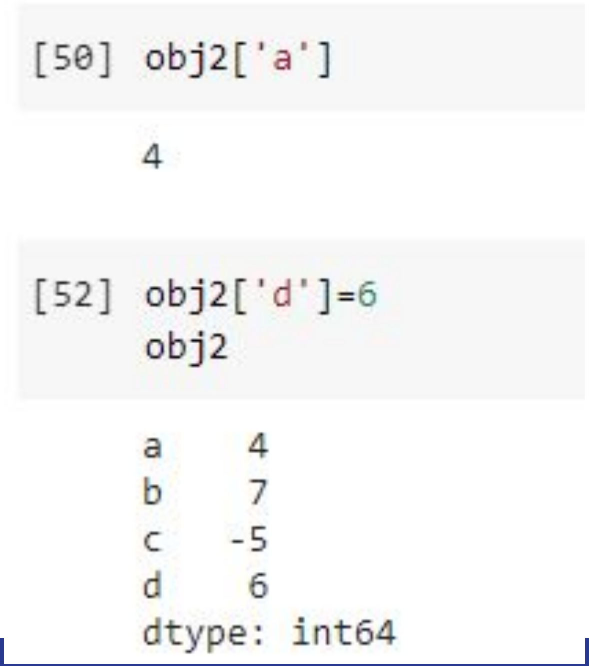


```
[49] obj2=pd.Series([4,7,-5,3],index=['a','b','c','d'])
     obj2
```

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

Series indexing

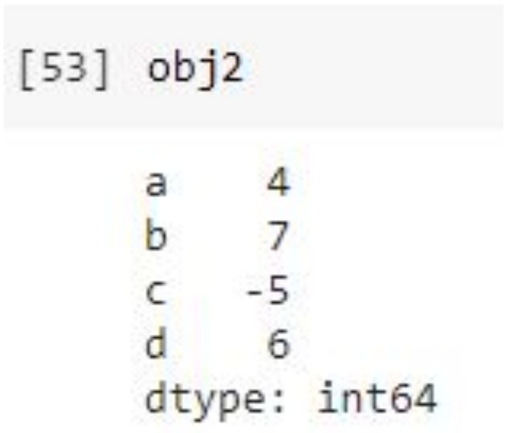
Compared with a regular NumPy array, you can use values in the index when selecting single values or a set of values:



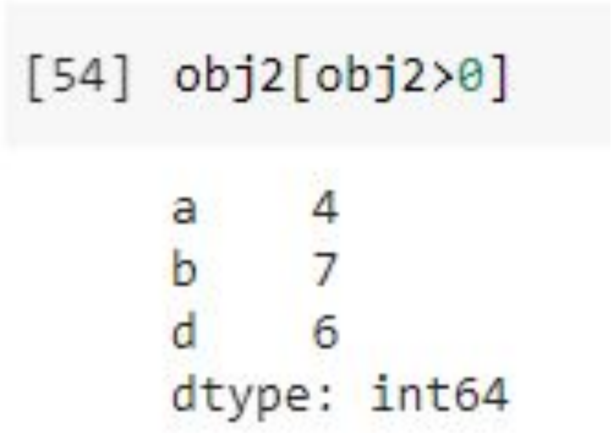
```
[50] obj2['a']

     4

[52] obj2['d']=6
     obj2

     a    4
     b    7
     c   -5
     d    6
     dtype: int64
```

Series indexing

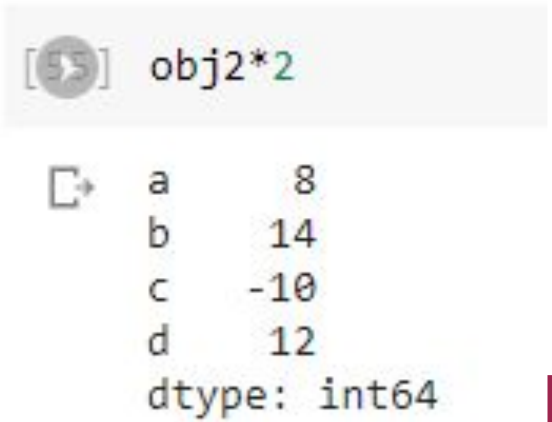
NumPy array operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link



```
[53] obj2

a    4
b    7
c   -5
d    6
dtype: int64
```



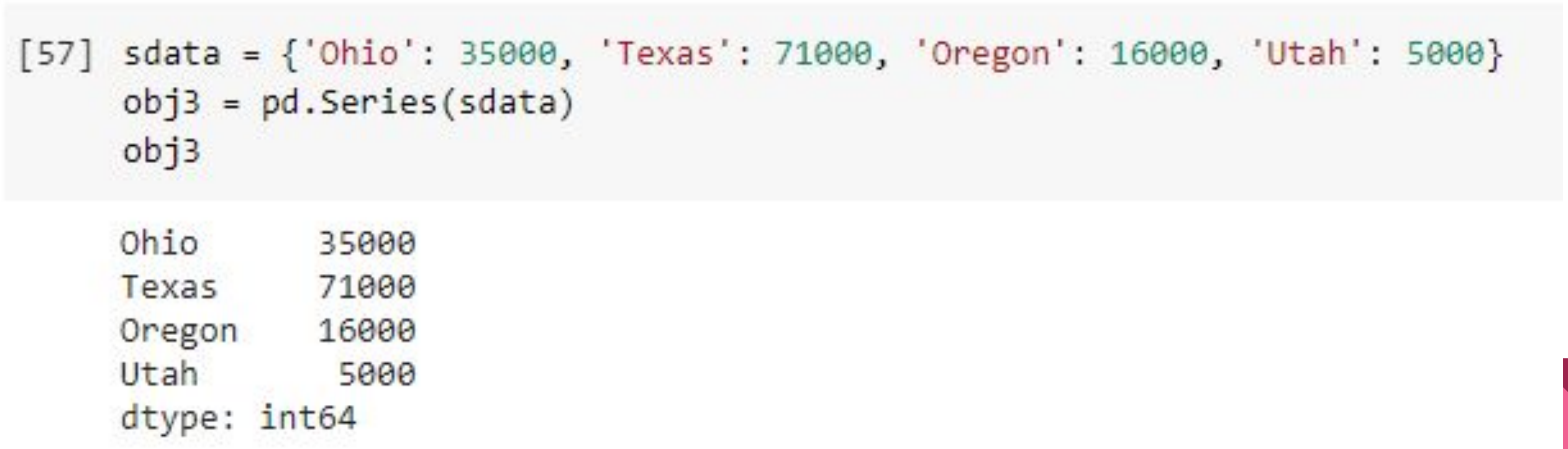
```
[54] obj2[obj2>0]

a    4
b    7
d    6
dtype: int64
```



```
[55] obj2*2

a     8
b    14
c   -10
d    12
dtype: int64
```

Series from dict

Should you have data contained in a Python dict, you can create a Series from it by passing the dict:



```
[57] sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
     obj3 = pd.Series(sdata)
     obj3

     Ohio      35000
     Texas     71000
     Oregon    16000
     Utah       5000
     dtype: int64
```

DataFrame

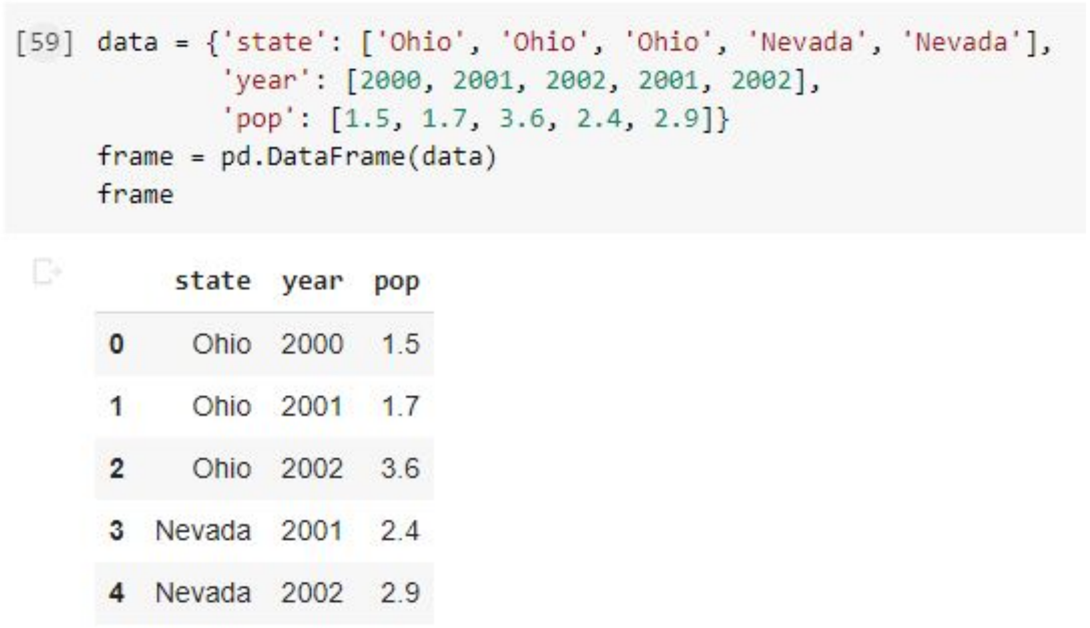
A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.).

The DataFrame has both a row and column index;

It can be thought of as a dict of Series (one for all sharing the same index).

Dataframe representation

There are numerous ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays

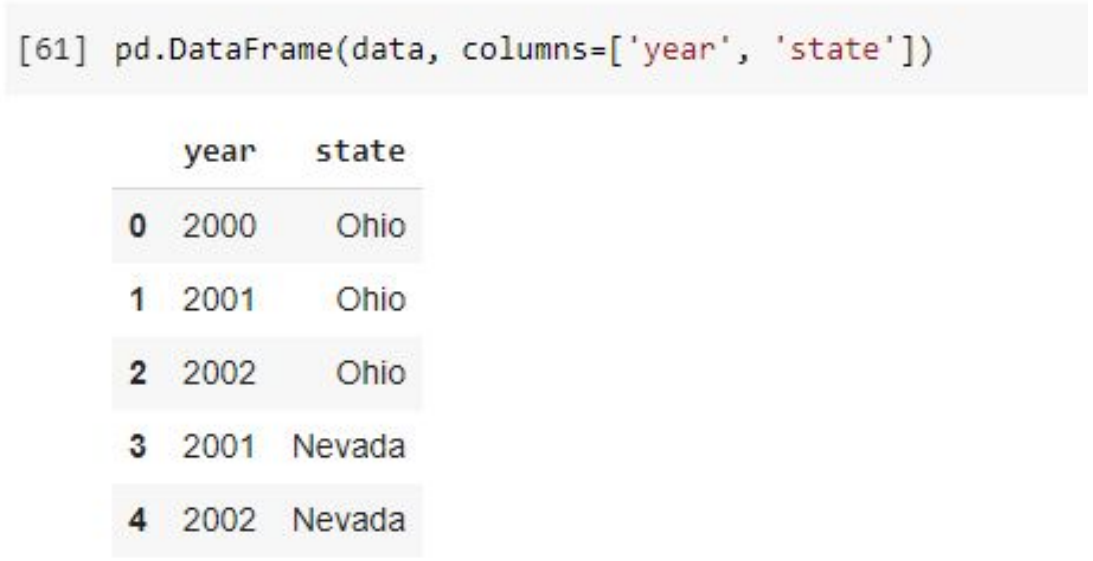


```
[59] data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
             'year': [2000, 2001, 2002, 2001, 2002],
             'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
     frame = pd.DataFrame(data)
     frame
```

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

Dataframes

If you specify a sequence of columns, the DataFrame's columns will be exactly what you pass:

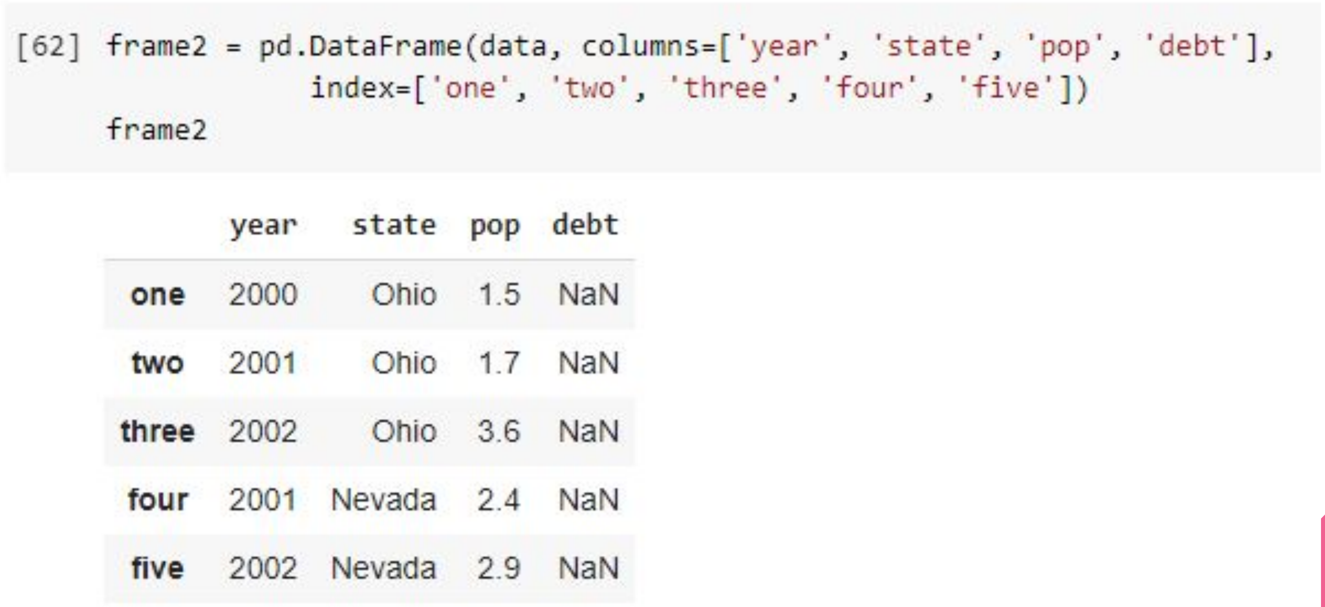


```
[61] pd.DataFrame(data, columns=['year', 'state'])
```

	year	state
0	2000	Ohio
1	2001	Ohio
2	2002	Ohio
3	2001	Nevada
4	2002	Nevada

Dataframes

As with Series, if you pass a column that isn't contained in data , it will appear with NA values in the result:

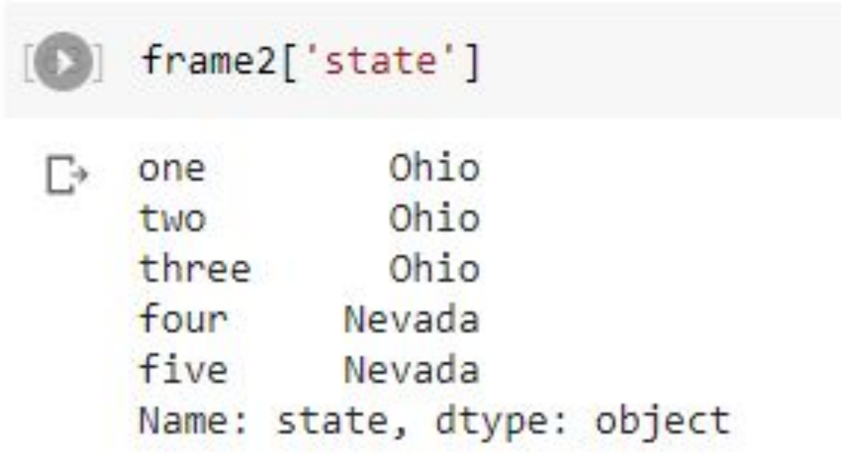


```
[62] frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                          index=['one', 'two', 'three', 'four', 'five'])
     frame2
```

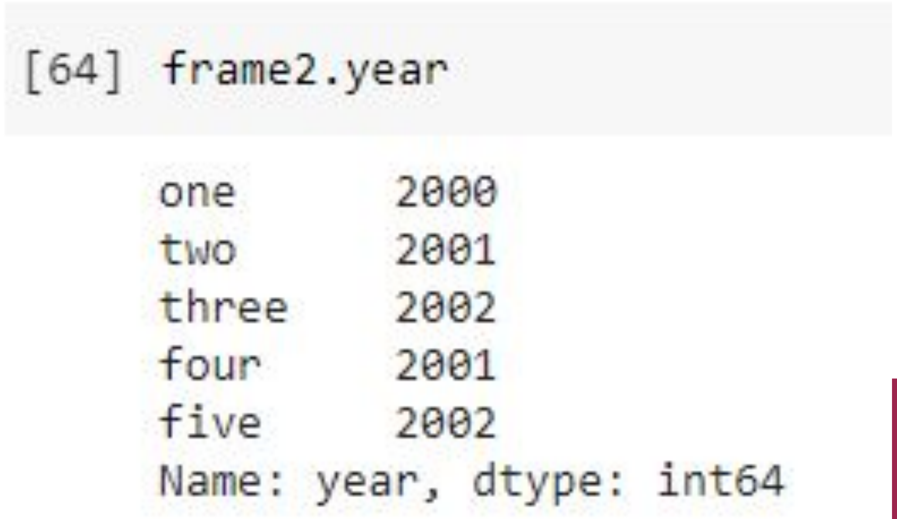
	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN

Dataframe columns

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:



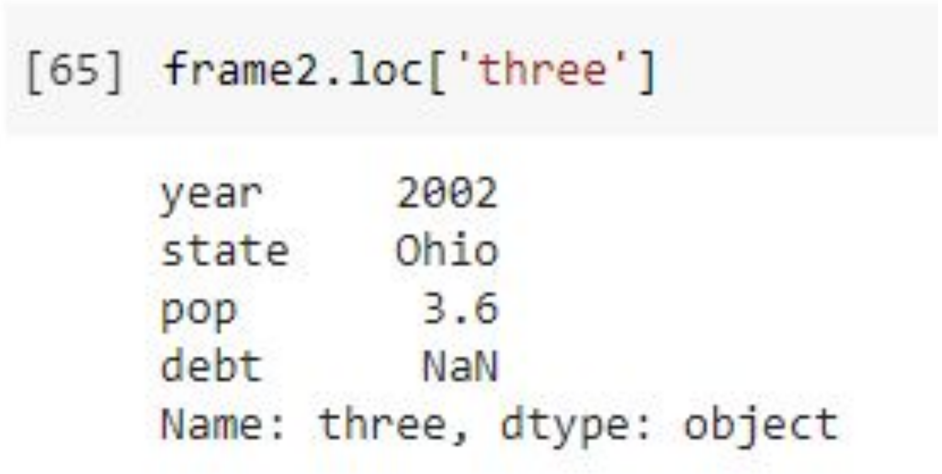
```
[ ] frame2['state']

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



```
[64] frame2.year

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

Dataframe rows

Rows can also be retrieved by position or name

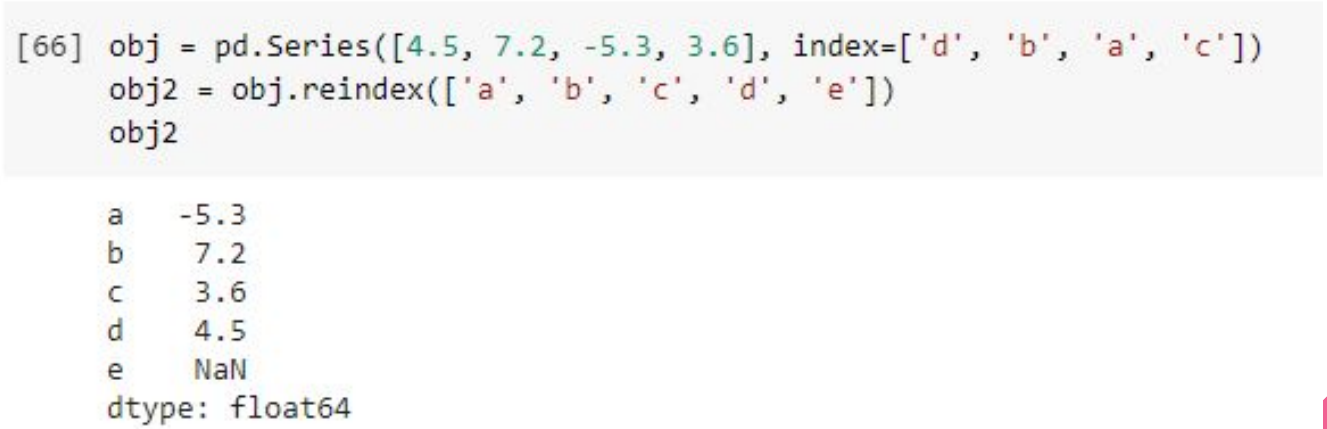


```
[65] frame2.loc['three']
```

```
year       2002
state      Ohio
pop         3.6
debt        NaN
Name: three, dtype: object
```

Essential Functionality: reindexing

A critical method on pandas objects is reindex , which means to create a new object with the data conformed to a new index. Consider a simple example from above:

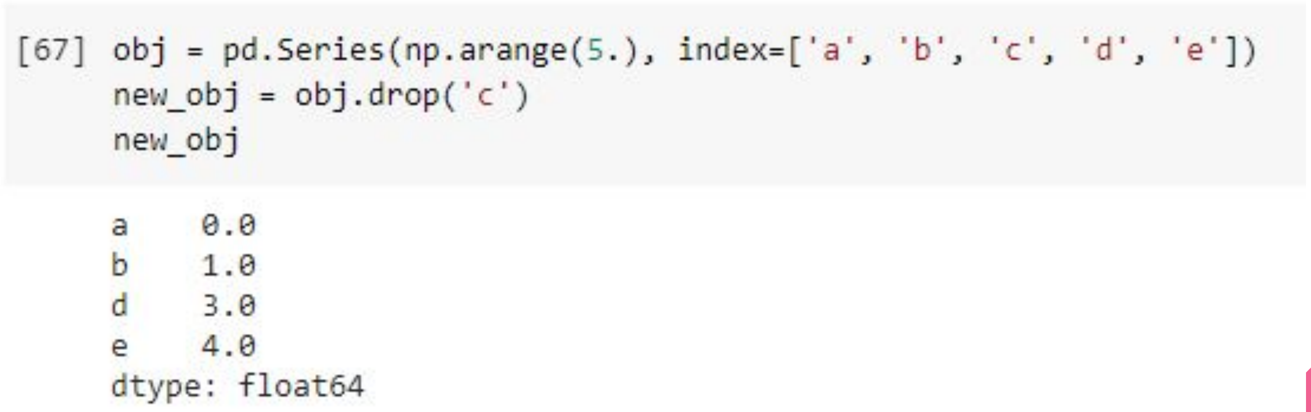


```
[66] obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
     obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
     obj2
```

```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

Dropping entries from an axis

Dropping one or more entries from an axis is easy if you have an index array or list without those entries. As that can require a bit of munging and set logic, the drop method will return a new object with the indicated value or values deleted from an axis:

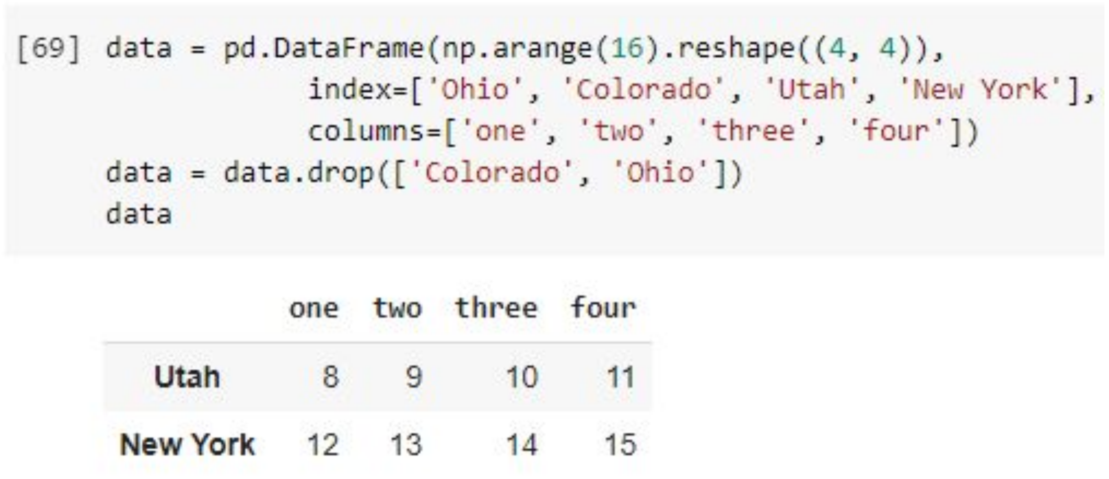


```
[67] obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
     new_obj = obj.drop('c')
     new_obj
```

```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

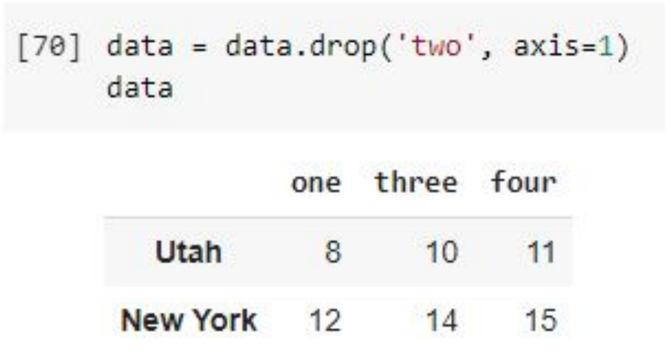
Dropping entries from an axis

With DataFrame, index values can be deleted from either axis:



```
[69] data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                         index=['Ohio', 'Colorado', 'Utah', 'New York'],
                         columns=['one', 'two', 'three', 'four'])
     data = data.drop(['Colorado', 'Ohio'])
     data
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

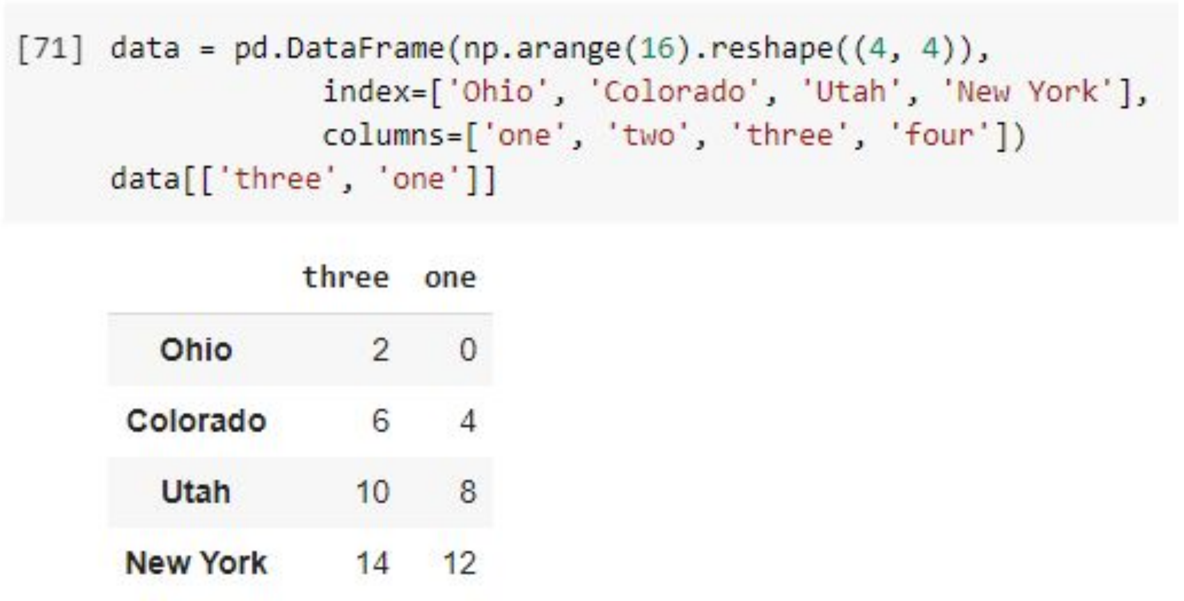


```
[70] data = data.drop('two', axis=1)
     data
```

	one	three	four
Utah	8	10	11
New York	12	14	15

Filtering in Dataframes

Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence

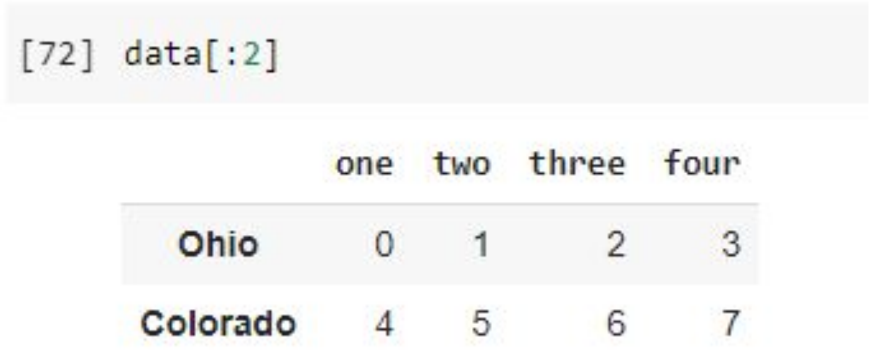


```
[71] data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                         index=['Ohio', 'Colorado', 'Utah', 'New York'],
                         columns=['one', 'two', 'three', 'four'])
     data[['three', 'one']]
```

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

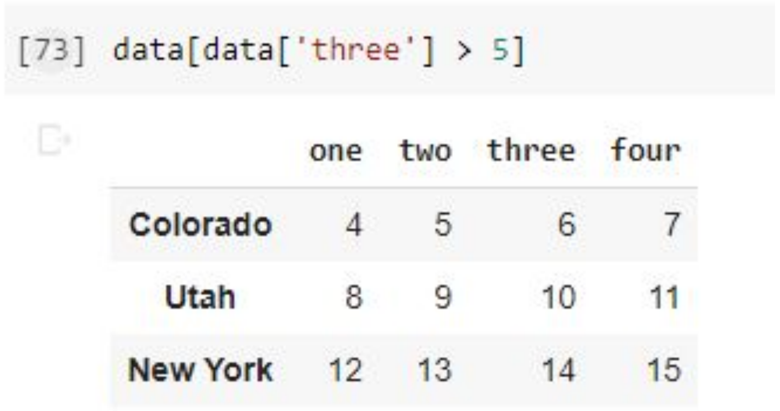
Filtering: special cases

Selecting rows by slicing or a boolean array:



```
[72] data[:2]
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

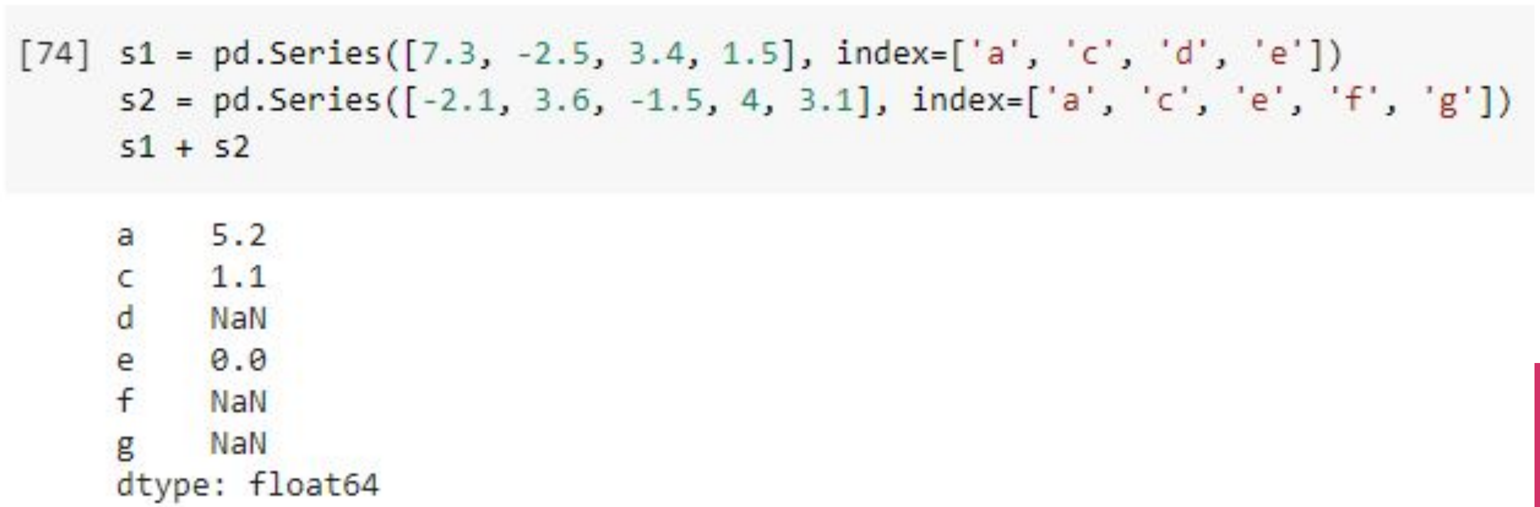


```
[73] data[data['three'] > 5]
```

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Arithmetic and data alignment

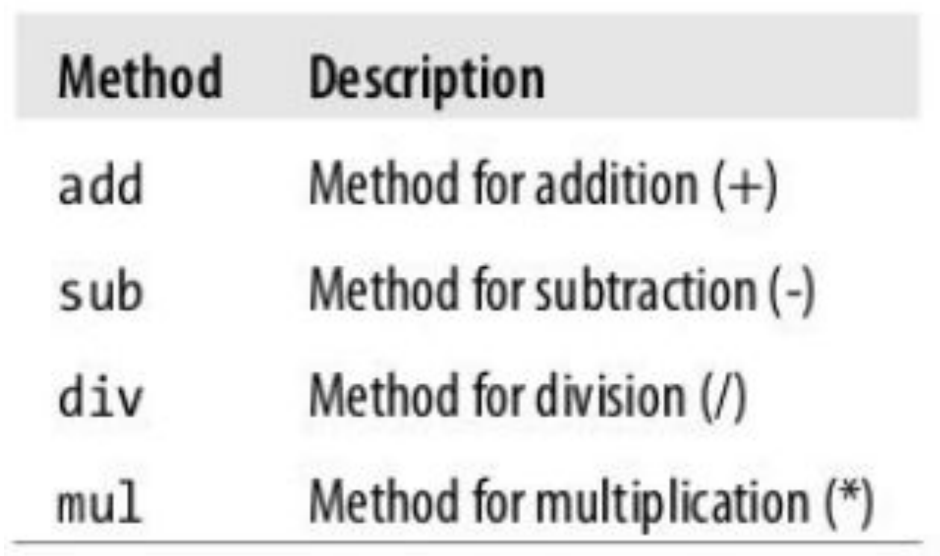
One of the most important pandas features is the behavior of arithmetic between objects with different indexes. When adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs.



```
[74] s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
     s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
     s1 + s2
```

```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

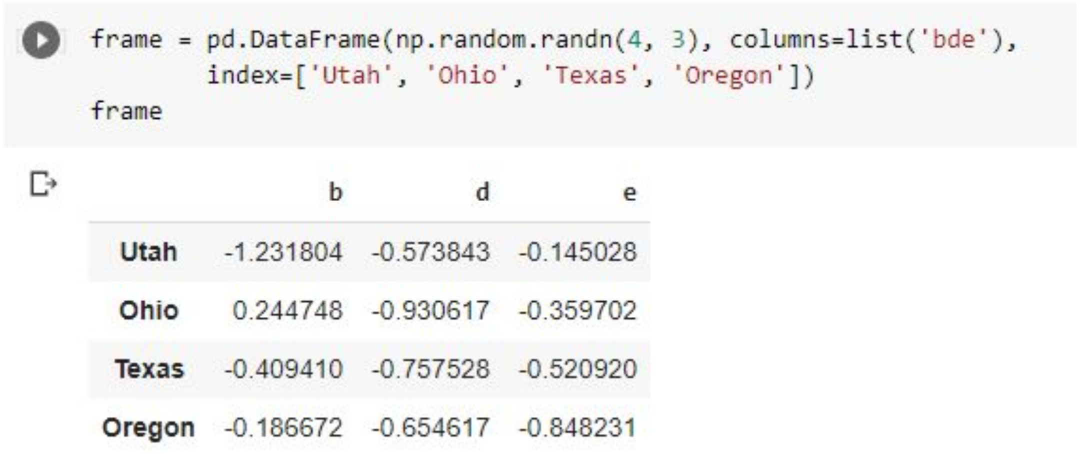
Arithmetic methods



Method	Description
`add`	Method for addition (+)
`sub`	Method for subtraction (-)
`div`	Method for division (/)
`mul`	Method for multiplication (*)

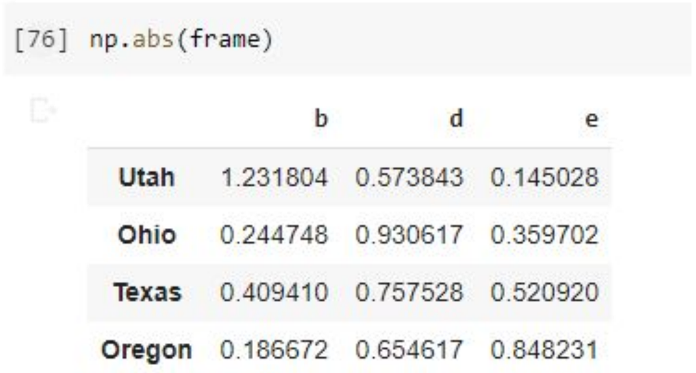
Function application and mapping

NumPy ufuncs (element-wise array methods) work fine with pandas objects.



```
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

	b	d	e
Utah	-1.231804	-0.573843	-0.145028
Ohio	0.244748	-0.930617	-0.359702
Texas	-0.409410	-0.757528	-0.520920
Oregon	-0.186672	-0.654617	-0.848231

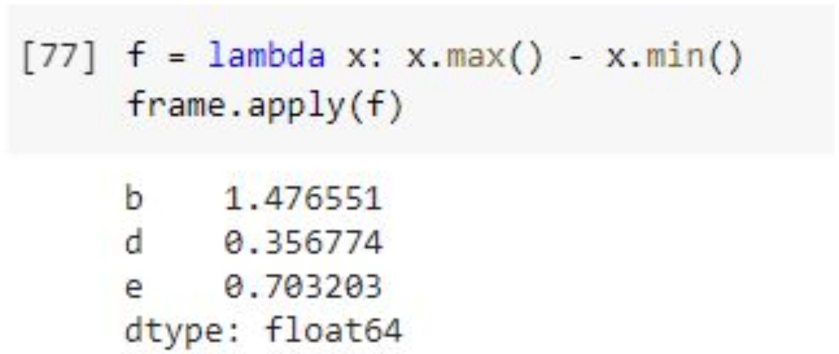


```
[76] np.abs(frame)
```

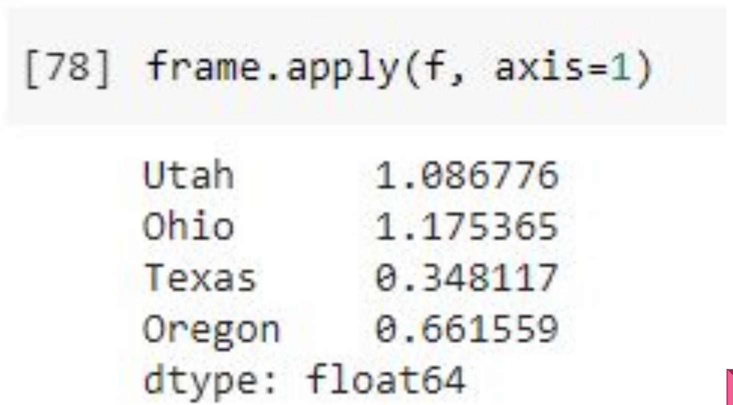
	b	d	e
Utah	1.231804	0.573843	0.145028
Ohio	0.244748	0.930617	0.359702
Texas	0.409410	0.757528	0.520920
Oregon	0.186672	0.654617	0.848231

lambda

Another frequent operation is applying a function on 1D arrays to each column or row. DataFrame’s apply method does exactly this:



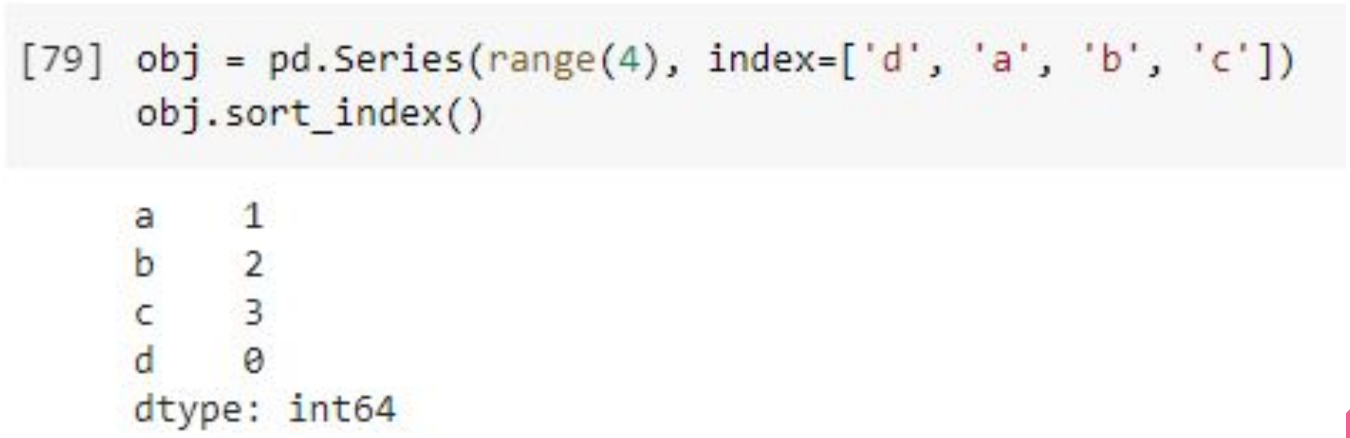
```
[77] f = lambda x: x.max() - x.min()
     frame.apply(f)

     b     1.476551
     d     0.356774
     e     0.703203
     dtype: float64
```



```
[78]  frame.apply(f, axis=1)

      Utah       1.086776
      Ohio       1.175365
      Texas      0.348117
      Oregon     0.661559
      dtype: float64
```

Sorting and ranking

Sorting a data set by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the sort_index method, which returns a new, sorted object:

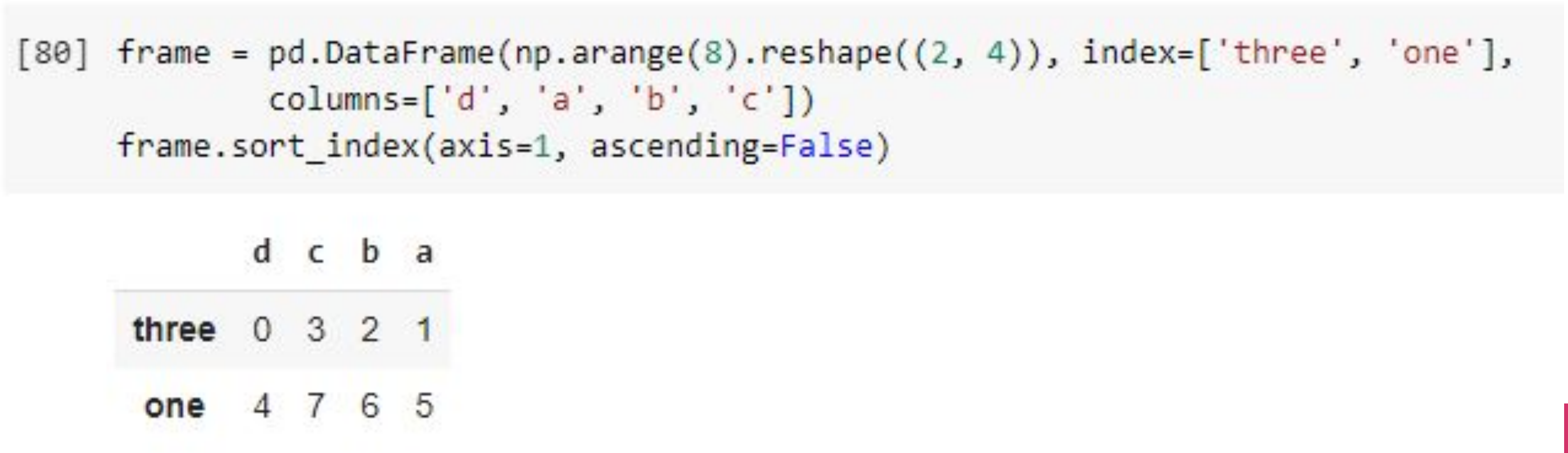


```
[79] obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
     obj.sort_index()
```

```
a    1
b    2
c    3
d    0
dtype: int64
```

Sorting and ranking

With a DataFrame, you can sort by index on either axis. The data is sorted in ascending order by default, but can be sorted in descending order, too:

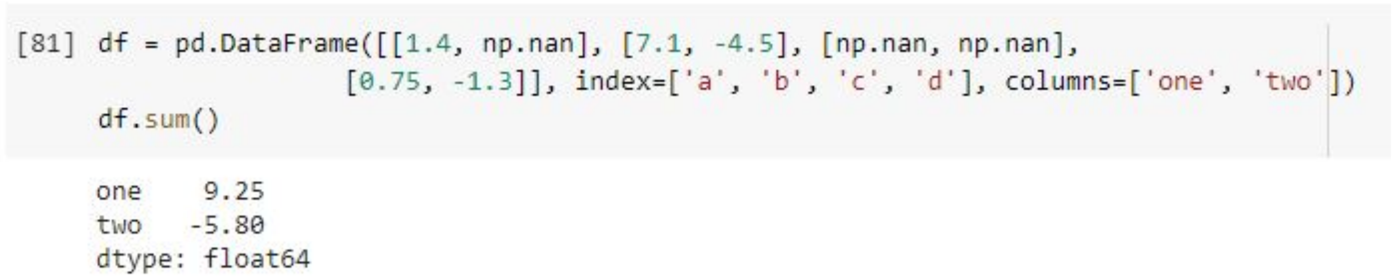


```
[80] frame = pd.DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
                          columns=['d', 'a', 'b', 'c'])
     frame.sort_index(axis=1, ascending=False)
```

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

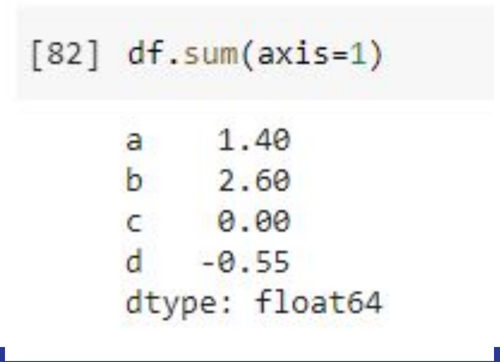
Summarizing

pandas objects are equipped with a set of common mathematical and statistical methods.



```
[81] df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5], [np.nan, np.nan],
                        [0.75, -1.3]], index=['a', 'b', 'c', 'd'], columns=['one', 'two'])
     df.sum()
```

```
one     9.25
two    -5.80
dtype: float64
```

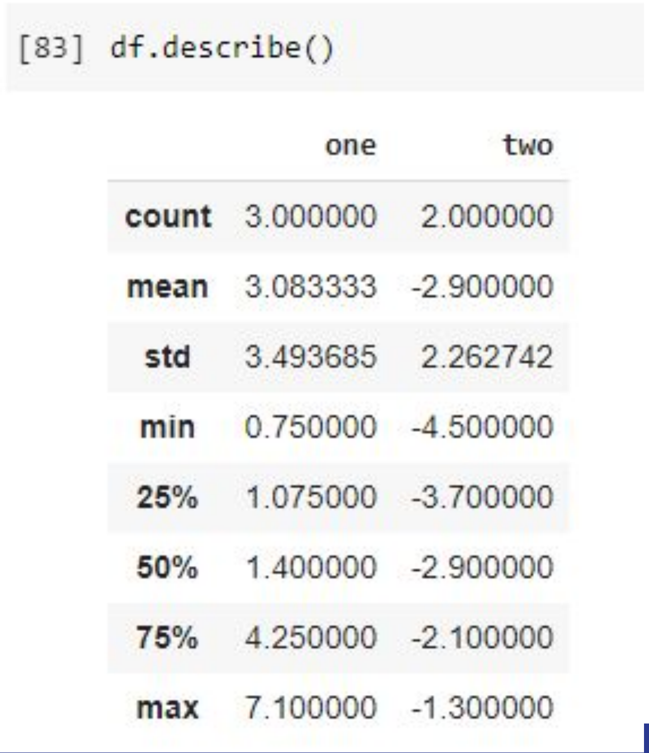


```
[82] df.sum(axis=1)
```

```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

describe

describe is one such example, producing multiple summary statistics in one shot:



```
[83] df.describe()
```

```
            one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

Data manipulation

Reading data

```
import pandas as pd
#Reading CSV data
activities = pd.read_csv("data/activities.csv")
#Reading HTML data and parsing all <table> tags
raw_data = pd.read_html("sdu_registration.xls",header=0, index_col=0)
#Reading Tab separated values data
table = pd.read_table(“data/counts.tsv”)
```

Viewing data

```
#viewing first 5 rows
df.head()
#viewing last 3 rows
df.tail(3)

#showing index and column parameters
df.index
df.columns
```

Saving data

```
df.to_csv('foo.csv')
df.to_table('foo.tsv')
df.to_html('foo.html')

#and more other functions
```

Conclusion

Numpy https://docs.scipy.org/doc/numpy/user/index.html

Pandas
http://pandas.pydata.org/pandas-docs/version/0.23/index.html

Pandas cheat sheet -
https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf

Pandas series from Kevin Markham -
ttps://github.com/justmarkham/pandas-videos