Search the docs ...

NumPy: the absolute basics for beginners

Welcome to the absolute beginner's guide to NumPy! If you have comments or suggestions, please don't hesitate to reach out!

Welcome to NumPy!

NumPy (**Numerical Python**) is an open source Python library that's used in almost every field of science and engineering. It's the universal standard for working with numerical data in Python, and it's at the core of the scientific Python and PyData ecosystems. NumPy users include everyone from beginning coders to experienced researchers doing state-of-the-art scientific and industrial research and development. The NumPy API is used extensively in Pandas, SciPy, Matplotlib, scikit-learn, scikit-image and most other data science and scientific Python packages.

The NumPy library contains multidimensional array and matrix data structures (you'll find more information about this in later sections). It provides **ndarray**, a homogeneous n-dimensional array object, with methods to efficiently operate on it. NumPy can be used to perform a wide variety of mathematical operations on arrays. It adds powerful data structures to Python that guarantee efficient calculations with arrays and matrices and it supplies an enormous library of high-level mathematical functions that operate on these arrays and matrices.

Learn more about NumPy here!

Installing NumPy

To install NumPy, we strongly recommend using a scientific Python distribution. If you're looking for the full instructions for installing NumPy on your operating system, you can find all of the details here.

If you already have Python, you can install NumPy with:

```
conda install numpy
```

or

```
pip install numpy
```

If you don’t have Python yet, you might want to consider using Anaconda. It’s the easiest way to get started. The good thing about getting this distribution is the fact that you don’t need to worry too much about separately installing NumPy or any of the major packages that you’ll be using for your data analyses, like pandas, Scikit-Learn, etc.

You can find all of the installation details in the Installation section at SciPy.

How to import NumPy

Any time you want to use a package or library in your code, you first need to make it accessible.

In order to start using NumPy and all of the functions available in NumPy, you’ll need to import it. This can be easily done with this import statement:

```
import numpy as np
```

(We shorten `numpy` to `np` in order to save time and also to keep code standardized so that anyone working with your code can easily understand and run it.)

Reading the example code

If you aren’t already comfortable with reading tutorials that contain a lot of code, you might not know how to interpret a code block that looks like this:

```
>>> a = np.arange(6)
>>> a2 = a[np.newaxis, :]
>>> a2.shape
(1, 6)
```

If you aren’t familiar with this style, it’s very easy to understand. If you see `>>>`, you’re looking at **input**, or the code that you would enter. Everything that doesn’t have `>>>` in front of it is **output**, or the results of running your code. This is the style you see when you run `python` on the command line, but if you’re using IPython, you might see a different style.

What’s the difference between a Python list and a NumPy array?

NumPy gives you an enormous range of fast and efficient ways of creating arrays and manipulating numerical data inside them. While a Python list can contain different data types within a single list, all of the elements in a NumPy array should be homogeneous. The mathematical operations that are meant to be performed on arrays would be extremely inefficient if the arrays weren’t homogeneous.

Why use NumPy?

NumPy arrays are faster and more compact than Python lists. An array consumes less memory and is convenient to use. NumPy uses much less memory to store data and it provides a mechanism of specifying the data types. This allows the code to be optimized even further.

What is an array?

An array is a central data structure of the NumPy library. An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. It has a grid of elements that can be indexed in various ways. The elements are all of the same type, referred to as the array `dtype`.

An array can be indexed by a tuple of nonnegative integers, by booleans, by another array, or by integers. The `rank` of the array is the number of dimensions. The `shape` of the array is a tuple of integers giving the size of the array along each dimension.

One way we can initialize NumPy arrays is from Python lists, using nested lists for two- or higher-dimensional data.

For example:

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

or:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

We can access the elements in the array using square brackets. When you're accessing elements, remember that indexing in NumPy starts at 0. That means that if you want to access the first element in your array, you'll be accessing element "0".

```
>>> print(a[0])
[1 2 3 4]
```

More information about arrays

This section covers `1D array`, `2D array`, `ndarray`, `vector`, `matrix`

You might occasionally hear an array referred to as a "ndarray," which is shorthand for "N-dimensional array." An N-dimensional array is simply an array with any number of dimensions. You might also hear **1-D**, or one-dimensional array, **2-D**, or two-dimensional array, and so on. The NumPy `ndarray` class is used to represent both matrices and vectors. A **vector** is an array with a single dimension (there's no difference between row and column vectors), while a **matrix** refers to an array with two dimensions. For **3-D** or higher dimensional arrays, the term **tensor** is also commonly used.

What are the attributes of an array?

An array is usually a fixed-size container of items of the same type and size. The number of dimensions and items in an array is defined by its shape. The shape of an array is a tuple of non-negative integers that specify the sizes of each dimension.

In NumPy, dimensions are called **axes**. This means that if you have a 2D array that looks like this:

```
[[0., 0., 0.],
 [1., 1., 1.]]
```

Your array has 2 axes. The first axis has a length of 2 and the second axis has a length of 3.

Just like in other Python container objects, the contents of an array can be accessed and modified by indexing or slicing the array. Unlike the typical container objects, different arrays can share the same data, so changes made on one array might be visible in another.

Array **attributes** reflect information intrinsic to the array itself. If you need to get, or even set, properties of an array without creating a new array, you can often access an array through its attributes.

Read more about array attributes here and learn about array objects here.

How to create a basic array

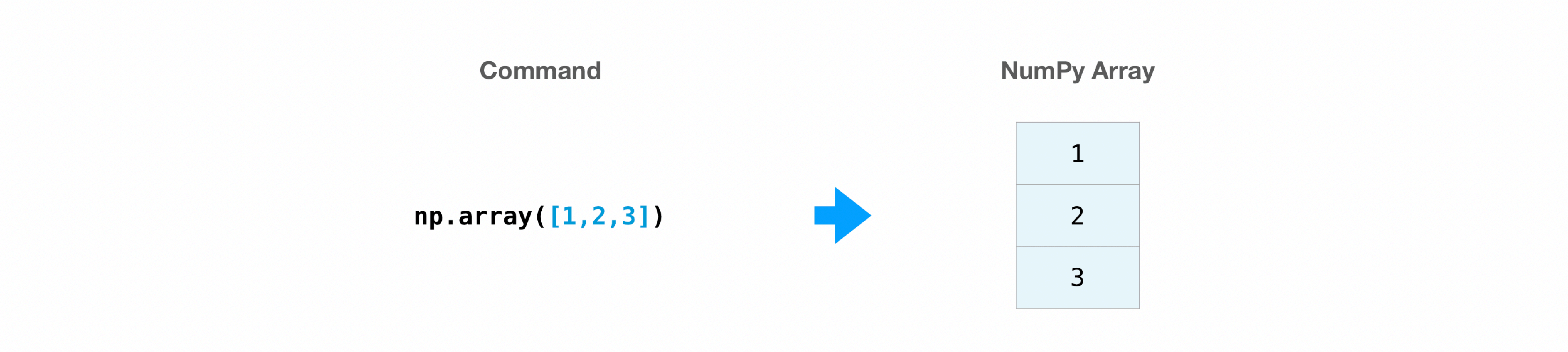
This section covers `np.array()`, `np.zeros()`, `np.ones()`, `np.empty()`, `np.arange()`, `np.linspace()`, `dtype`

To create a NumPy array, you can use the function `np.array()`.

All you need to do to create a simple array is pass a list to it. If you choose to, you can also specify the type of data in your list. You can find more information about data types here.

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
```

You can visualize your array this way:



Be aware that these visualizations are meant to simplify ideas and give you a basic understanding of NumPy concepts and mechanics. Arrays and array operations are much more complicated than are captured here!

Besides creating an array from a sequence of elements, you can easily create an array filled with `0`’s:

```
>>> np.zeros(2)
array([0., 0.])
```

Or an array filled with `1`’s:

```
>>> np.ones(2)
array([1., 1.])
```

Or even an empty array! The function `empty` creates an array whose initial content is random and depends on the state of the memory. The reason to use `empty` over `zeros` (or something similar) is speed - just make sure to fill every element afterwards!

```
>>> # Create an empty array with 2 elements
>>> np.empty(2)
array([ 3.14, 42.  ])  # may vary
```

You can create an array with a range of elements:

```
>>> np.arange(4)
array([0, 1, 2, 3])
```

And even an array that contains a range of evenly spaced intervals. To do this, you will specify the **first number**, **last number**, and the **step size**.

```
>>> np.arange(2, 9, 2)
array([2, 4, 6, 8])
```

You can also use `np.linspace()` to create an array with values that are spaced linearly in a specified interval:

```
>>> np.linspace(0, 10, num=5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

Specifying your data type

While the default data type is floating point (`np.float64`), you can explicitly specify which data type you want using the `dtype` keyword.

```
>>> x = np.ones(2, dtype=np.int64)
>>> x
array([1, 1])
```

Learn more about creating arrays here

Adding, removing, and sorting elements

This section covers `np.sort()`, `np.concatenate()`

Sorting an element is simple with `np.sort()`. You can specify the axis, kind, and order when you call the function.

If you start with this array:

```
>>> arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])
```

You can quickly sort the numbers in ascending order with:

```
>>> np.sort(arr)
array([1, 2, 3, 4, 5, 6, 7, 8])
```

In addition to sort, which returns a sorted copy of an array, you can use:

- **`argsort`**, which is an indirect sort along a specified axis,
- **`lexsort`**, which is an indirect stable sort on multiple keys,
- **`searchsorted`**, which will find elements in a sorted array, and
- **`partition`**, which is a partial sort.

To read more about sorting an array, see: **`sort`**.

If you start with these arrays:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([5, 6, 7, 8])
```

You can concatenate them with `np.concatenate()`.

```
>>> np.concatenate((a, b))
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Or, if you start with these arrays:

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

You can concatenate them with:

```
>>> np.concatenate((x, y), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In order to remove elements from an array, it’s simple to use indexing to select the elements that you want to keep.

To read more about concatenate, see: **`concatenate`**.

How do you know the shape and size of an array?

This section covers `ndarray.ndim`, `ndarray.size`, `ndarray.shape`

`ndarray.ndim` will tell you the number of axes, or dimensions, of the array.

`ndarray.size` will tell you the total number of elements of the array. This is the *product* of the elements of the array's shape.

`ndarray.shape` will display a tuple of integers that indicate the number of elements stored along each dimension of the array. If, for example, you have a 2-D array with 2 rows and 3 columns, the shape of your array is `(2, 3)`.

For example, if you create this array:

```
>>> array_example = np.array([[[0, 1, 2, 3],
...                            [4, 5, 6, 7]],
...
...                           [[0, 1, 2, 3],
...                            [4, 5, 6, 7]],
...
...                           [[0 ,1 ,2, 3],
...                            [4, 5, 6, 7]]])
```

To find the number of dimensions of the array, run:

```
>>> array_example.ndim
3
```

To find the total number of elements in the array, run:

```
>>> array_example.size
24
```

And to find the shape of your array, run:

```
>>> array_example.shape
(3, 2, 4)
```

Can you reshape an array?

This section covers `arr.reshape()`

Yes!

Using `arr.reshape()` will give a new shape to an array without changing the data. Just remember that when you use the reshape method, the array you want to produce needs to have the same number of elements as the original array. If you start with an array with 12 elements, you'll need to make sure that your new array also has a total of 12 elements.

If you start with this array:

```
>>> a = np.arange(6)
>>> print(a)
[0 1 2 3 4 5]
```

You can use `reshape()` to reshape your array. For example, you can reshape this array to an array with three rows and two columns:

```
>>> b = a.reshape(3, 2)
>>> print(b)
[[0 1]
 [2 3]
 [4 5]]
```

With `np.reshape`, you can specify a few optional parameters:

```
>>> numpy.reshape(a, newshape=(1, 6), order='C')
array([[0, 1, 2, 3, 4, 5]])
```

`a` is the array to be reshaped.

`newshape` is the new shape you want. You can specify an integer or a tuple of integers. If you specify an integer, the result will be an array of that length. The shape should be compatible with the original shape.

`order:` `C` means to read/write the elements using C-like index order, `F` means to read/write the elements using Fortran-like index order, `A` means to read/write the elements in Fortran-like index order if a is Fortran contiguous in memory, C-like order otherwise. (This is an optional parameter and doesn't need to be specified.)

If you want to learn more about C and Fortran order, you can read more about the internal organization of NumPy arrays here. Essentially, C and Fortran orders have to do with how indices correspond to the order the array is stored in memory. In Fortran, when moving through the elements of a two-dimensional array as it is stored in memory, the **first** index is the most rapidly varying index. As the first index moves to the next row as it changes, the matrix is stored one column at a time. This is why Fortran is thought of as a **Column-major language**. In C on the other hand, the **last** index changes the most rapidly. The matrix is stored by rows, making it a **Row-major language**. What you do for C or Fortran depends on whether it's more important to preserve the indexing convention or not reorder the data.

Learn more about shape manipulation here.

How to convert a 1D array into a 2D array (how to add a new axis to an array)

This section covers `np.newaxis`, `np.expand_dims`

You can use `np.newaxis` and `np.expand_dims` to increase the dimensions of your existing array.

Using `np.newaxis` will increase the dimensions of your array by one dimension when used once. This means that a **1D** array will become a **2D** array, a **2D** array will become a **3D** array, and so on.

For example, if you start with this array:

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

You can use `np.newaxis` to add a new axis:

```
>>> a2 = a[np.newaxis, :]
>>> a2.shape
(1, 6)
```

You can explicitly convert a 1D array with either a row vector or a column vector using `np.newaxis`. For example, you can convert a 1D array to a row vector by inserting an axis along the first dimension:

```
>>> row_vector = a[np.newaxis, :]
>>> row_vector.shape
(1, 6)
```

Or, for a column vector, you can insert an axis along the second dimension:

```
>>> col_vector = a[:, np.newaxis]
>>> col_vector.shape
(6, 1)
```

You can also expand an array by inserting a new axis at a specified position with `np.expand_dims`.

For example, if you start with this array:

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

You can use `np.expand_dims` to add an axis at index position 1 with:

```
>>> b = np.expand_dims(a, axis=1)
>>> b.shape
(6, 1)
```

You can add an axis at index position 0 with:

```
>>> c = np.expand_dims(a, axis=0)
>>> c.shape
(1, 6)
```

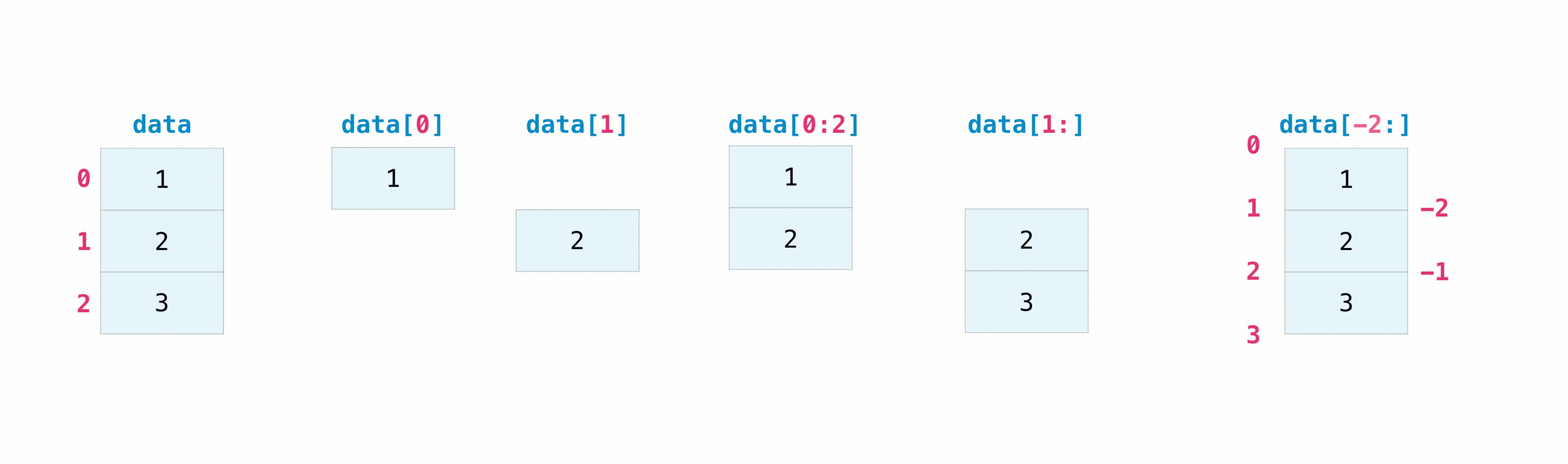
Find more information about newaxis here and `expand_dims` at **`expand_dims`**.

Indexing and slicing

You can index and slice NumPy arrays in the same ways you can slice Python lists.

```
>>> data = np.array([1, 2, 3])

>>> data[1]
2
>>> data[0:2]
array([1, 2])
>>> data[1:]
array([2, 3])
>>> data[-2:]
array([2, 3])
```

You can visualize it this way:



You may want to take a section of your array or specific array elements to use in further analysis or additional operations. To do that, you'll need to subset, slice, and/or index your arrays.

If you want to select values from your array that fulfill certain conditions, it's straightforward with NumPy.

For example, if you start with this array:

```
>>> a = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can easily print all of the values in the array that are less than 5.

```
>>> print(a[a < 5])
[1 2 3 4]
```

You can also select, for example, numbers that are equal to or greater than 5, and use that condition to index an array.

```
>>> five_up = (a >= 5)
>>> print(a[five_up])
[ 5  6  7  8  9 10 11 12]
```

You can select elements that are divisible by 2:

```
>>> divisible_by_2 = a[a%2==0]
>>> print(divisible_by_2)
[ 2  4  6  8 10 12]
```

Or you can select elements that satisfy two conditions using the `&` and `|` operators:

```
>>> c = a[(a > 2) & (a < 11)]
>>> print(c)
[ 3  4  5  6  7  8  9 10]
```

You can also make use of the logical operators **&** and **|** in order to return boolean values that specify whether or not the values in an array fulfill a certain condition. This can be useful with arrays that contain names or other categorical values.

```
>>> five_up = (a > 5) | (a == 5)
>>> print(five_up)
[[False False False False]
 [ True  True  True  True]
 [ True  True  True True]]
```

You can also use `np.nonzero()` to select elements or indices from an array.

Starting with this array:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can use `np.nonzero()` to print the indices of elements that are, for example, less than 5:

```
>>> b = np.nonzero(a < 5)
>>> print(b)
(array([0, 0, 0, 0]), array([0, 1, 2, 3]))
```

In this example, a tuple of arrays was returned: one for each dimension. The first array represents the row indices where these values are found, and the second array represents the column indices where the values are found.

If you want to generate a list of coordinates where the elements exist, you can zip the arrays, iterate over the list of coordinates, and print them. For example:

```
>>> list_of_coordinates= list(zip(b[0], b[1]))

>>> for coord in list_of_coordinates:
...     print(coord)
(0, 0)
(0, 1)
(0, 2)
(0, 3)
```

You can also use `np.nonzero()` to print the elements in an array that are less than 5 with:

```
>>> print(a[b])
[1 2 3 4]
```

If the element you're looking for doesn't exist in the array, then the returned array of indices will be empty. For example:

```
>>> not_there = np.nonzero(a == 42)
>>> print(not_there)
(array([], dtype=int64), array([], dtype=int64))
```

Learn more about indexing and slicing here and here.

Read more about using the nonzero function at: **`nonzero`**.

How to create an array from existing data

This section covers `slicing and indexing`, `np.vstack()`, `np.hstack()`, `np.hsplit()`, `.view()`, `copy()`

You can easily use create a new array from a section of an existing array.

Let's say you have this array:

```
>>> a = np.array([1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

You can create a new array from a section of your array any time by specifying where you want to slice your array.

```
>>> arr1 = a[3:8]
>>> arr1
array([4, 5, 6, 7, 8])
```

Here, you grabbed a section of your array from index position 3 through index position 8.

You can also stack two existing arrays, both vertically and horizontally. Let's say you have two arrays, `a1` and `a2`:

```
>>> a1 = np.array([[1, 1],
...                [2, 2]])

>>> a2 = np.array([[3, 3],
...                [4, 4]])
```

You can stack them vertically with `vstack`:

```
>>> np.vstack((a1, a2))
array([[1, 1],
       [2, 2],
       [3, 3],
       [4, 4]])
```

Or stack them horizontally with `hstack`:

```
>>> np.hstack((a1, a2))
array([[1, 1, 3, 3],
       [2, 2, 4, 4]])
```

You can split an array into several smaller arrays using `hsplit`. You can specify either the number of equally shaped arrays to return or the columns *after* which the division should occur.

Let’s say you have this array:

```
>>> x = np.arange(1, 25).reshape(2, 12)
>>> x
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]])
```

If you wanted to split this array into three equally shaped arrays, you would run:

```
>>> np.hsplit(x, 3)
[array([[1,  2,  3,  4],
        [13, 14, 15, 16]]), array([[ 5,  6,  7,  8],
        [17, 18, 19, 20]]), array([[ 9, 10, 11, 12],
        [21, 22, 23, 24]])]
```

If you wanted to split your array after the third and fourth column, you’d run:

```
>>> np.hsplit(x, (3, 4))
[array([[1, 2, 3],
        [13, 14, 15]]), array([[ 4],
        [16]]), array([[ 5, 6, 7, 8, 9, 10, 11, 12],
        [17, 18, 19, 20, 21, 22, 23, 24]])]
```

Learn more about stacking and splitting arrays here.

You can use the `view` method to create a new array object that looks at the same data as the original array (a *shallow copy*).

Views are an important NumPy concept! NumPy functions, as well as operations like indexing and slicing, will return views whenever possible. This saves memory and is faster (no copy of the data has to be made). However it’s important to be aware of this - modifying data in a view also modifies the original array!

Let’s say you create this array:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

Now we create an array `b1` by slicing `a` and modify the first element of `b1`. This will modify the corresponding element in `a` as well!

```
>>> b1 = a[0, :]
>>> b1
array([1, 2, 3, 4])
>>> b1[0] = 99
>>> b1
array([99,  2,  3,  4])
>>> a
array([[99,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Using the `copy` method will make a complete copy of the array and its data (a *deep copy*). To use this on your array, you could run:

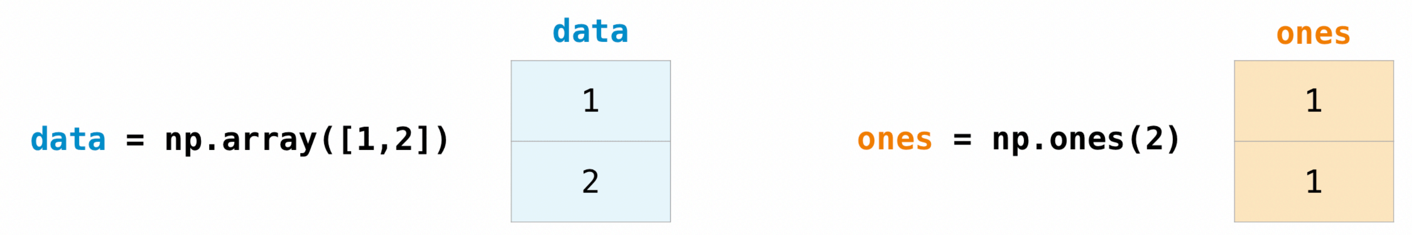
```
>>> b2 = a.copy()
```

Learn more about copies and views here.

Basic array operations

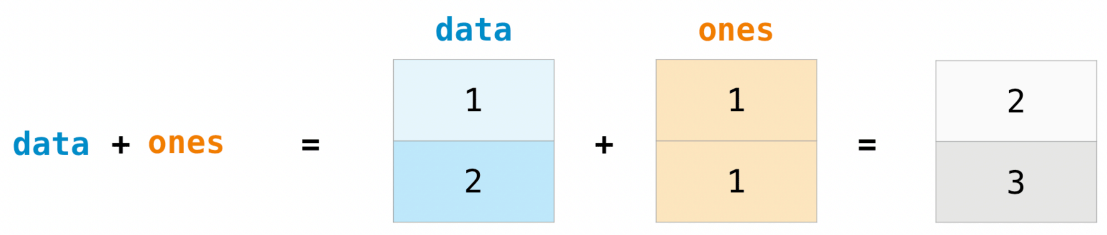
This section covers addition, subtraction, multiplication, division, and more

Once you've created your arrays, you can start to work with them. Let's say, for example, that you've created two arrays, one called "data" and one called "ones"



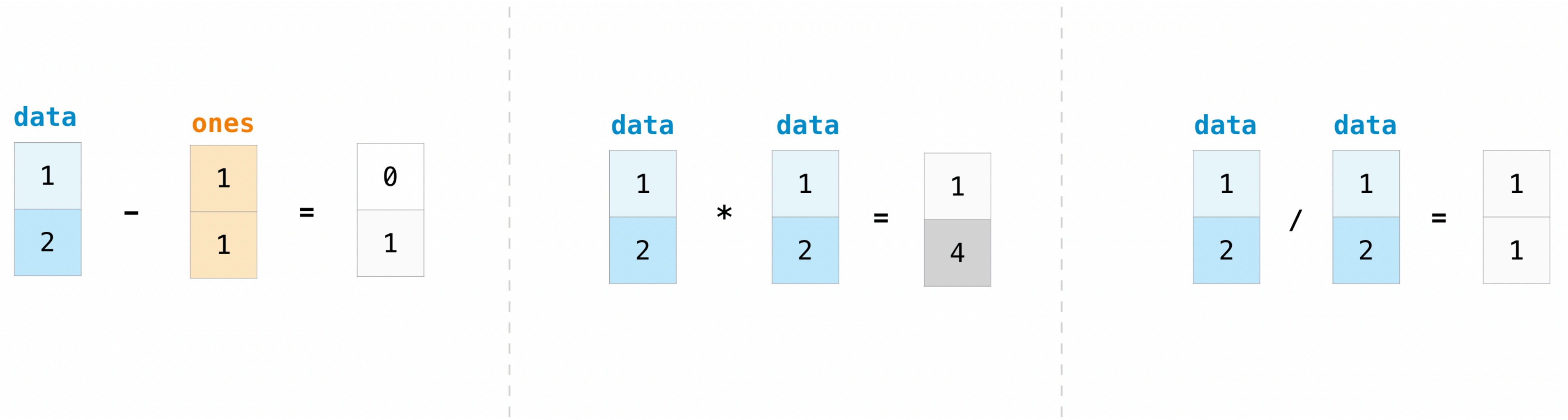
You can add the arrays together with the plus sign.

```
>>> data = np.array([1, 2])
>>> ones = np.ones(2, dtype=int)
>>> data + ones
array([2, 3])
```



You can, of course, do more than just addition!

```
>>> data - ones
array([0, 1])
>>> data * data
array([1, 4])
>>> data / data
array([1., 1.])
```



Basic operations are simple with NumPy. If you want to find the sum of the elements in an array, you'd use `sum()`. This works for 1D arrays, 2D arrays, and arrays in higher dimensions.

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

>>> a.sum()
10
```

To add the rows or the columns in a 2D array, you would specify the axis.

If you start with this array:

```
>>> b = np.array([[1, 1], [2, 2]])
```

You can sum the rows with:

```
>>> b.sum(axis=0)
array([3, 3])
```

You can sum the columns with:

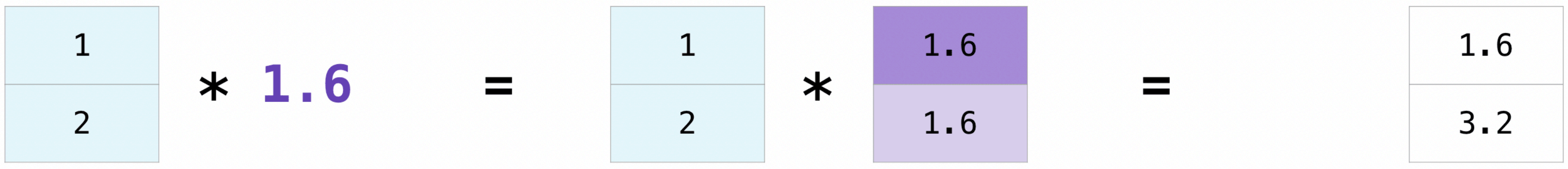
```
>>> b.sum(axis=1)
array([2, 4])
```

Learn more about basic operations here.

Broadcasting

There are times when you might want to carry out an operation between an array and a single number (also called *an operation between a vector and a scalar*) or between arrays of two different sizes. For example, your array (we'll call it "data") might contain information about distance in miles but you want to convert the information to kilometers. You can perform this operation with:

```
>>> data = np.array([1.0, 2.0])
>>> data * 1.6
array([1.6, 3.2])
```



NumPy understands that the multiplication should happen with each cell. That concept is called **broadcasting**. Broadcasting is a mechanism that allows NumPy to perform operations on arrays of different shapes. The dimensions of your array must be compatible, for example, when the dimensions of both arrays are equal or when one of them is 1. If the dimensions are not compatible, you will get a `ValueError`.

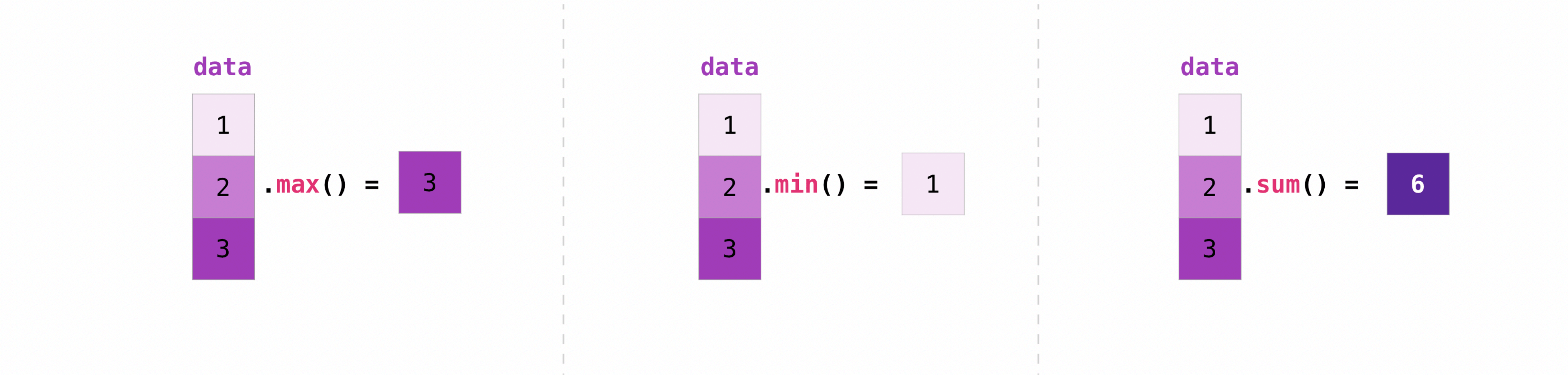
Learn more about broadcasting here.

More useful array operations

This section covers maximum, minimum, sum, mean, product, standard deviation, and more

NumPy also performs aggregation functions. In addition to `min`, `max`, and `sum`, you can easily run `mean` to get the average, `prod` to get the result of multiplying the elements together, `std` to get the standard deviation, and more.

```
>>> data.max()
2.0
>>> data.min()
1.0
>>> data.sum()
3.0
```



Let’s start with this array, called “a”

```
>>> a = np.array([[0.45053314, 0.17296777, 0.34376245, 0.5510652],
...               [0.54627315, 0.05093587, 0.40067661, 0.55645993],
...               [0.12697628, 0.82485143, 0.26590556, 0.56917101]])
```

It’s very common to want to aggregate along a row or column. By default, every NumPy aggregation function will return the aggregate of the entire array. To find the sum or the minimum of the elements in your array, run:

```
>>> a.sum()
4.8595784
```

Or:

```
>>> a.min()
0.05093587
```

You can specify on which axis you want the aggregation function to be computed. For example, you can find the minimum value within each column by specifying `axis=0`.

```
>>> a.min(axis=0)
array([0.12697628, 0.05093587, 0.26590556, 0.5510652 ])
```

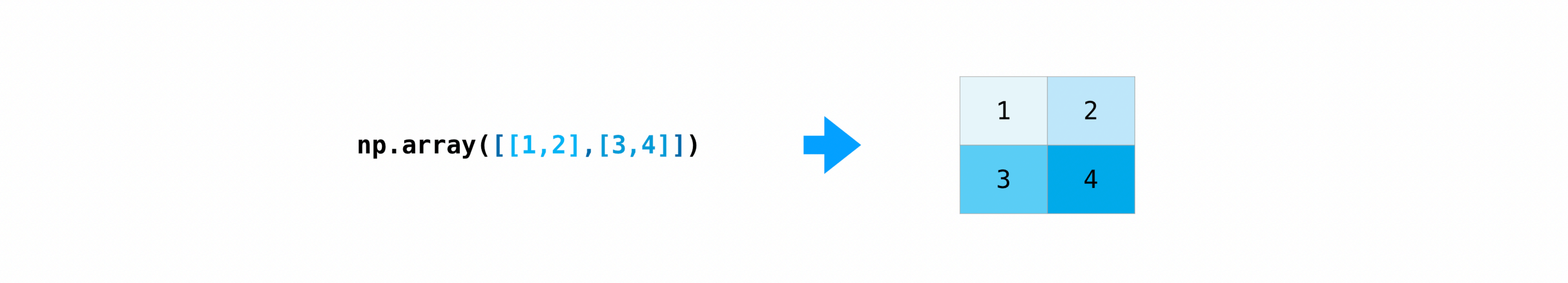
The four values listed above correspond to the number of columns in your array. With a four-column array, you will get four values as your result.

Read more about array methods here.

Creating matrices

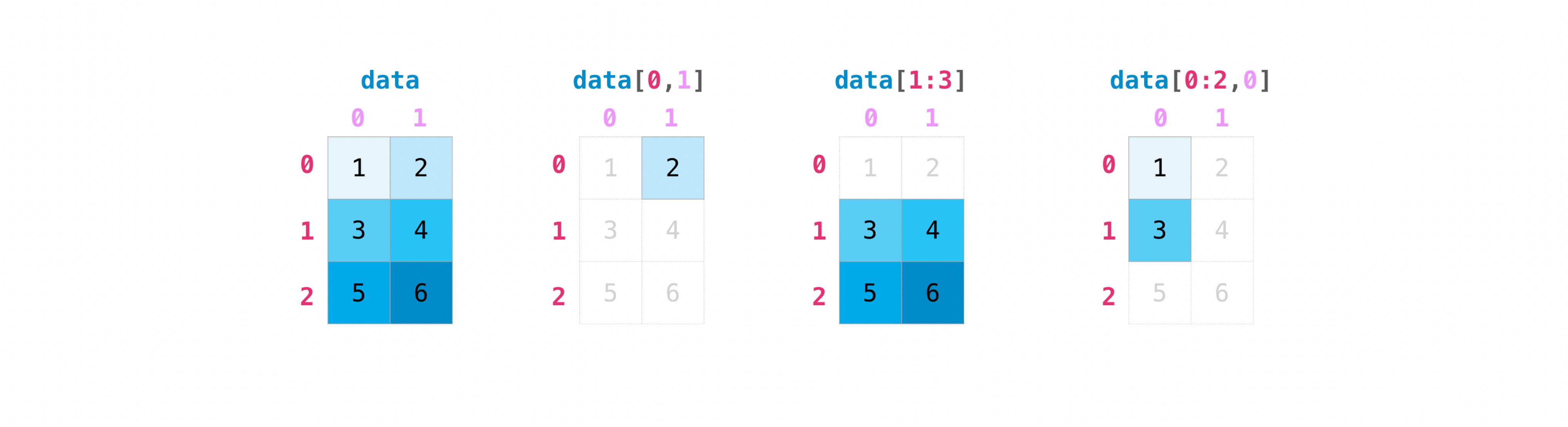
You can pass Python lists of lists to create a 2-D array (or “matrix”) to represent them in NumPy.

```
>>> data = np.array([[1, 2], [3, 4]])
>>> data
array([[1, 2],
       [3, 4]])
```



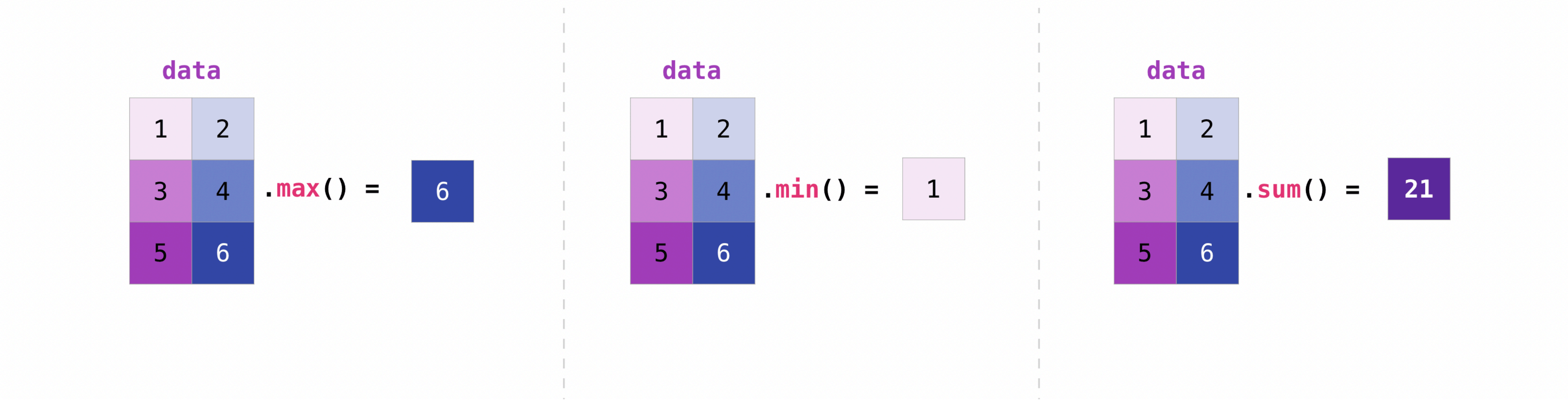
Indexing and slicing operations are useful when you're manipulating matrices:

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



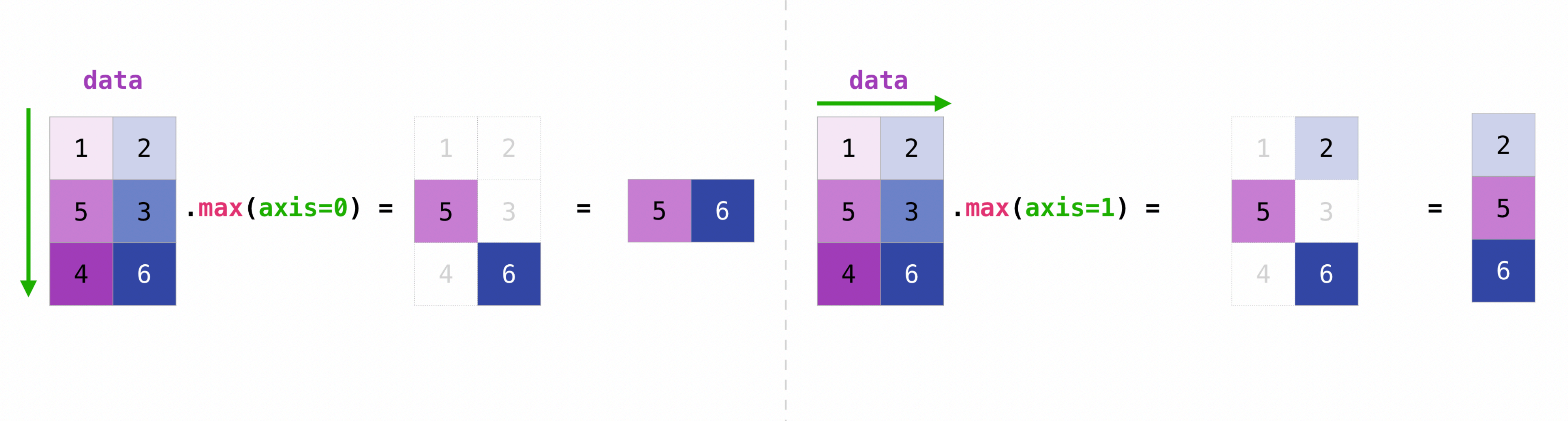
You can aggregate matrices the same way you aggregated vectors:

```
>>> data.max()
4
>>> data.min()
1
>>> data.sum()
10
```



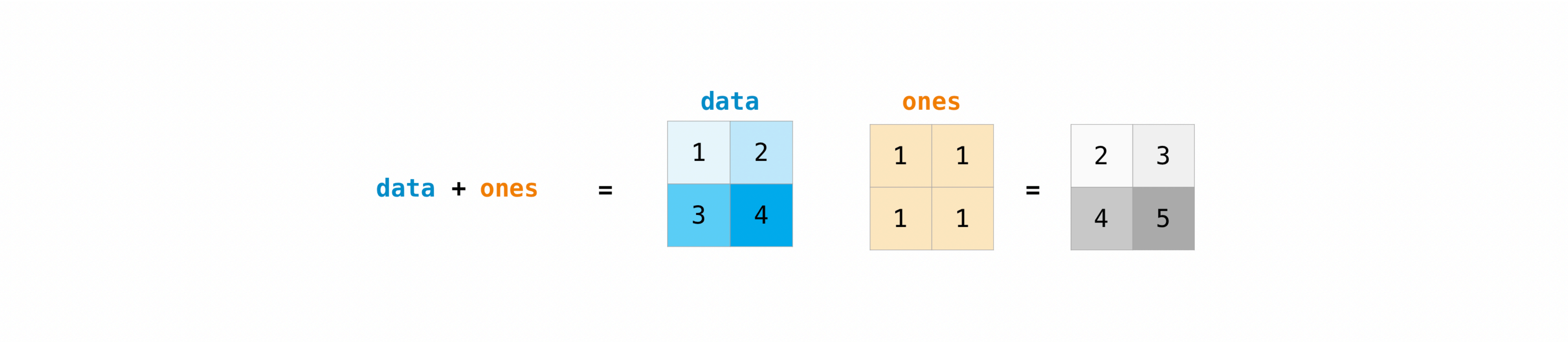
You can aggregate all the values in a matrix and you can aggregate them across columns or rows using the `axis` parameter:

```
>>> data.max(axis=0)
array([3, 4])
>>> data.max(axis=1)
array([2, 4])
```



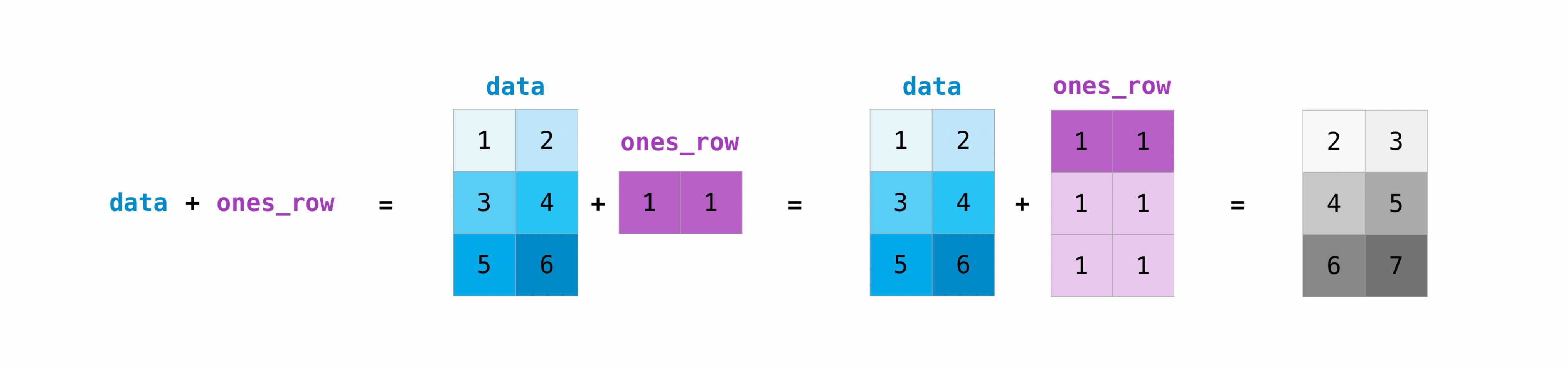
Once you've created your matrices, you can add and multiply them using arithmetic operators if you have two matrices that are the same size.

```
>>> data = np.array([[1, 2], [3, 4]])
>>> ones = np.array([[1, 1], [1, 1]])
>>> data + ones
array([[2, 3],
       [4, 5]])
```



You can do these arithmetic operations on matrices of different sizes, but only if one matrix has only one column or one row. In this case, NumPy will use its broadcast rules for the operation.

```
>>> data = np.array([[1, 2], [3, 4], [5, 6]])
>>> ones_row = np.array([[1, 1]])
>>> data + ones_row
array([[2, 3],
       [4, 5],
       [6, 7]])
```

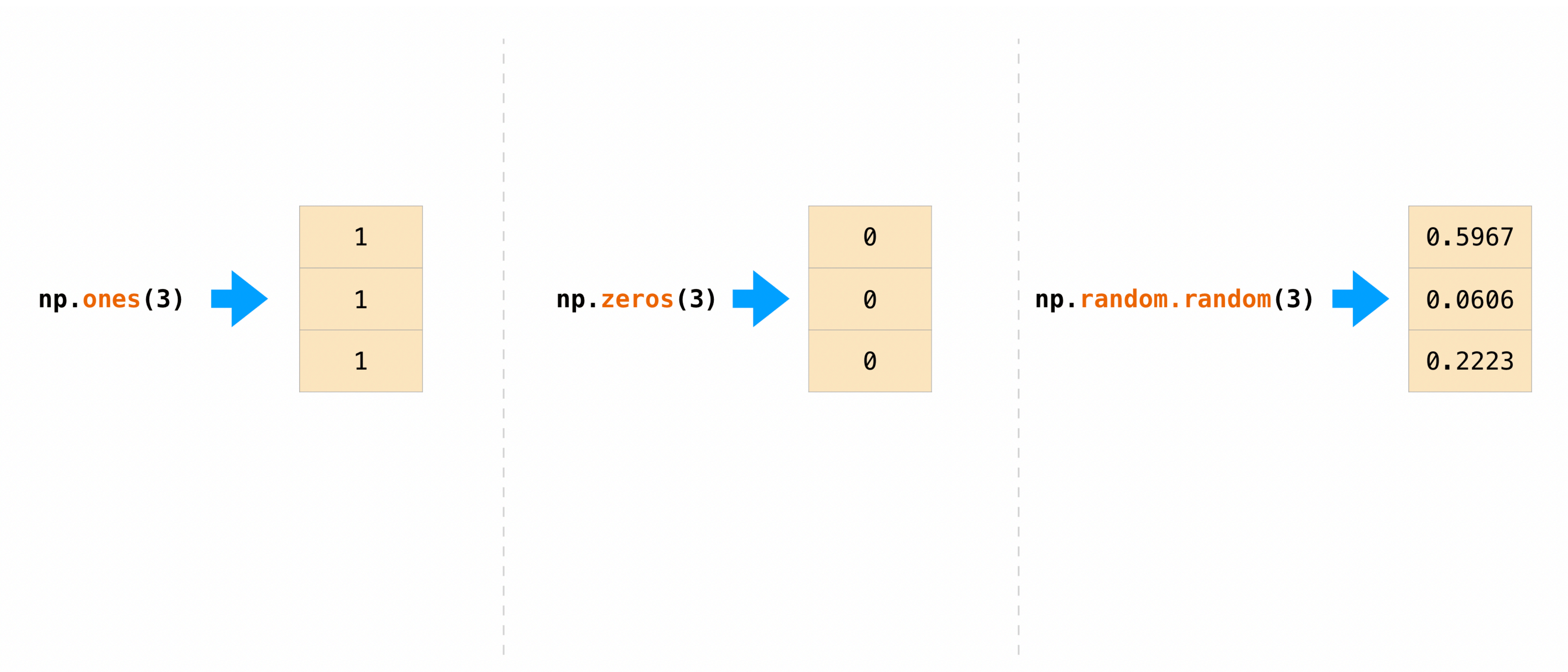


Be aware that when NumPy prints N-dimensional arrays, the last axis is looped over the fastest while the first axis is the slowest. For instance:

```
>>> np.ones((4, 3, 2))
array([[[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]]])
```

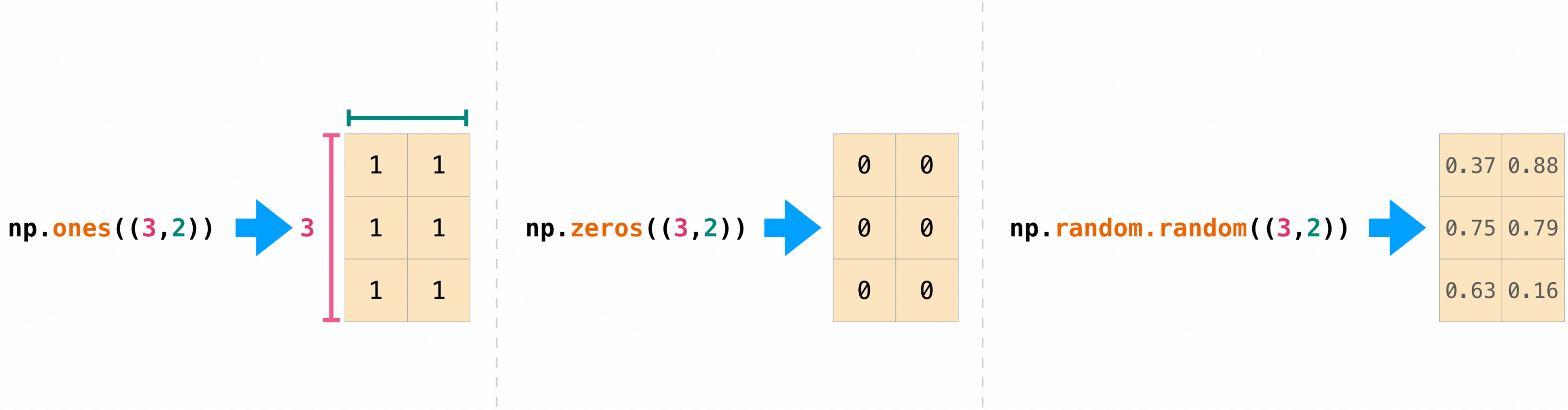
There are often instances where we want NumPy to initialize the values of an array. NumPy offers functions like `ones()` and `zeros()`, and the `random.Generator` class for random number generation for that. All you need to do is pass in the number of elements you want it to generate:

```
>>> np.ones(3)
array([1., 1., 1.])
>>> np.zeros(3)
array([0., 0., 0.])
# the simplest way to generate random numbers
>>> rng = np.random.default_rng(0)
>>> rng.random(3)
array([0.63696169, 0.26978671, 0.04097352])
```



You can also use `ones()`, `zeros()`, and `random()` to create a 2D array if you give them a tuple describing the dimensions of the matrix:

```
>>> np.ones((3, 2))
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
>>> np.zeros((3, 2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> rng.random((3, 2))
array([[0.01652764, 0.81327024],
       [0.91275558, 0.60663578],
       [0.72949656, 0.54362499]])  # may vary
```



Read more about creating arrays, filled with `0`'s, `1`'s, other values or uninitialized, at array creation routines.

Generating random numbers

The use of random number generation is an important part of the configuration and evaluation of many numerical and machine learning algorithms. Whether you need to randomly initialize weights in an artificial neural network, split data into random sets, or randomly shuffle your dataset, being able to generate random numbers (actually, repeatable pseudo-random numbers) is essential.

With `Generator.integers`, you can generate random integers from low (remember that this is inclusive with NumPy) to high (exclusive). You can set `endpoint=True` to make the high number inclusive.

You can generate a 2 x 4 array of random integers between 0 and 4 with:

```
>>> rng.integers(5, size=(2, 4))
array([[2, 1, 1, 0],
       [0, 0, 0, 4]])  # may vary
```

Read more about random number generation here.

How to get unique items and counts

This section covers `np.unique()`

You can find the unique elements in an array easily with `np.unique`.

For example, if you start with this array:

```
>>> a = np.array([11, 11, 12, 13, 14, 15, 16, 17, 12, 13, 11, 14, 18, 19, 20])
```

you can use `np.unique` to print the unique values in your array:

```
>>> unique_values = np.unique(a)
>>> print(unique_values)
[11 12 13 14 15 16 17 18 19 20]
```

To get the indices of unique values in a NumPy array (an array of first index positions of unique values in the array), just pass the `return_index` argument in `np.unique()` as well as your array.

```
>>> unique_values, indices_list = np.unique(a, return_index=True)
>>> print(indices_list)
[ 0  2  3  4  5  6  7 12 13 14]
```

You can pass the `return_counts` argument in `np.unique()` along with your array to get the frequency count of unique values in a NumPy array.

```
>>> unique_values, occurrence_count = np.unique(a, return_counts=True)
>>> print(occurrence_count)
[3 2 2 2 1 1 1 1 1 1]
```

This also works with 2D arrays! If you start with this array:

```
>>> a_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [1, 2, 3, 4]])
```

You can find unique values with:

```
>>> unique_values = np.unique(a_2d)
>>> print(unique_values)
[ 1  2  3  4  5  6  7  8  9 10 11 12]
```

If the axis argument isn't passed, your 2D array will be flattened.

If you want to get the unique rows or columns, make sure to pass the `axis` argument. To find the unique rows, specify `axis=0` and for columns, specify `axis=1`.

```
>>> unique_rows = np.unique(a_2d, axis=0)
>>> print(unique_rows)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

To get the unique rows, index position, and occurrence count, you can use:

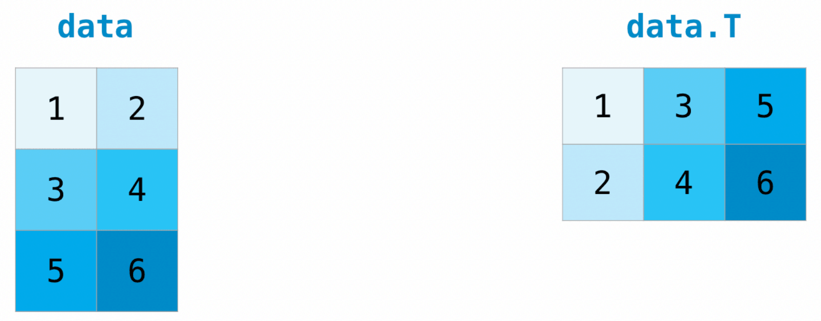
```
>>> unique_rows, indices, occurrence_count = np.unique(
...      a_2d, axis=0, return_counts=True, return_index=True)
>>> print(unique_rows)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
>>> print(indices)
[0 1 2]
>>> print(occurrence_count)
[2 1 1]
```

To learn more about finding the unique elements in an array, see **`unique`**.

Transposing and reshaping a matrix

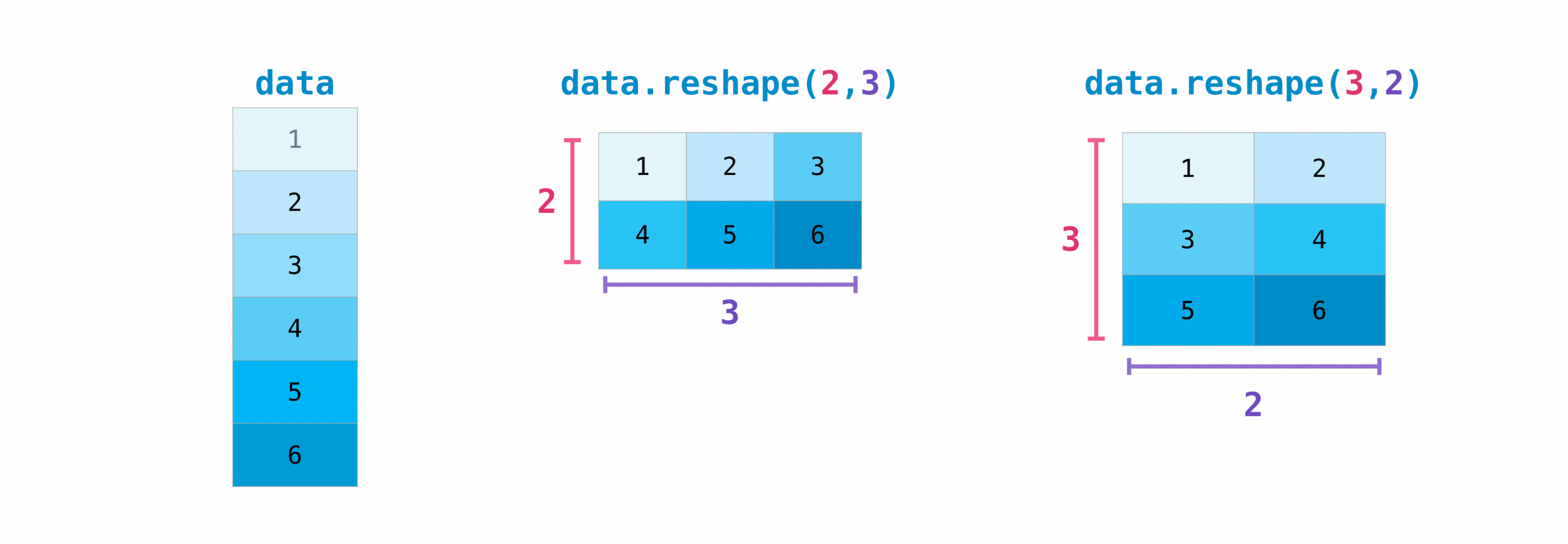
This section covers `arr.reshape()`, `arr.transpose()`, `arr.T`

It's common to need to transpose your matrices. NumPy arrays have the property `T` that allows you to transpose a matrix.



You may also need to switch the dimensions of a matrix. This can happen when, for example, you have a model that expects a certain input shape that is different from your dataset. This is where the `reshape` method can be useful. You simply need to pass in the new dimensions that you want for the matrix.

```
>>> data.reshape(2, 3)
array([[1, 2, 3],
       [4, 5, 6]])
>>> data.reshape(3, 2)
array([[1, 2],
       [3, 4],
       [5, 6]])
```



You can also use `.transpose()` to reverse or change the axes of an array according to the values you specify.

If you start with this array:

```
>>> arr = np.arange(6).reshape((2, 3))
>>> arr
array([[0, 1, 2],
       [3, 4, 5]])
```

You can transpose your array with `arr.transpose()`.

```
>>> arr.transpose()
array([[0, 3],
       [1, 4],
       [2, 5]])
```

You can also use `arr.T`:

```
>>> arr.T
array([[0, 3],
       [1, 4],
       [2, 5]])
```

To learn more about transposing and reshaping arrays, see **`transpose`** and **`reshape`**.

How to reverse an array

This section covers `np.flip()`

NumPy's `np.flip()` function allows you to flip, or reverse, the contents of an array along an axis. When using `np.flip()`, specify the array you would like to reverse and the axis. If you don't specify the axis, NumPy will reverse the contents along all of the axes of your input array.

Reversing a 1D array

If you begin with a 1D array like this one:

```
>>> arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
```

You can reverse it with:

```
>>> reversed_arr = np.flip(arr)
```

If you want to print your reversed array, you can run:

```
>>> print('Reversed Array: ', reversed_arr)
Reversed Array:  [8 7 6 5 4 3 2 1]
```

Reversing a 2D array

A 2D array works much the same way.

If you start with this array:

```
>>> arr_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can reverse the content in all of the rows and all of the columns with:

```
>>> reversed_arr = np.flip(arr_2d)
>>> print(reversed_arr)
[[12 11 10  9]
 [ 8  7  6  5]
 [ 4  3  2  1]]
```

You can easily reverse only the *rows* with:

```
>>> reversed_arr_rows = np.flip(arr_2d, axis=0)
>>> print(reversed_arr_rows)
[[ 9 10 11 12]
 [ 5  6  7  8]
 [ 1  2  3  4]]
```

Or reverse only the *columns* with:

```
>>> reversed_arr_columns = np.flip(arr_2d, axis=1)
>>> print(reversed_arr_columns)
[[ 4  3  2  1]
 [ 8  7  6  5]
 [12 11 10  9]]
```

You can also reverse the contents of only one column or row. For example, you can reverse the contents of the row at index position 1 (the second row):

```
>>> arr_2d[1] = np.flip(arr_2d[1])
>>> print(arr_2d)
[[ 1  2  3  4]
 [ 8  7  6  5]
 [ 9 10 11 12]]
```

You can also reverse the column at index position 1 (the second column):

```
>>> arr_2d[:,1] = np.flip(arr_2d[:,1])
>>> print(arr_2d)
[[ 1 10  3  4]
 [ 8  7  6  5]
 [ 9  2 11 12]]
```

Read more about reversing arrays at **flip**.

Reshaping and flattening multidimensional arrays

This section covers `.flatten()`, `ravel()`

There are two popular ways to flatten an array: `.flatten()` and `.ravel()`. The primary difference between the two is that the new array created using `ravel()` is actually a reference to the parent array (i.e., a “view”). This means that any changes to the new array will affect the parent array as well. Since `ravel` does not create a copy, it’s memory efficient.

If you start with this array:

```
>>> x = np.array([[1 , 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

You can use `flatten` to flatten your array into a 1D array.

```
>>> x.flatten()
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

When you use `flatten`, changes to your new array won’t change the parent array.

For example:

```
>>> a1 = x.flatten()
>>> a1[0] = 99
>>> print(x)  # Original array
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
>>> print(a1)  # New array
[99  2  3  4  5  6  7  8  9 10 11 12]
```

But when you use `ravel`, the changes you make to the new array will affect the parent array.

For example:

```
>>> a2 = x.ravel()
>>> a2[0] = 98
>>> print(x)  # Original array
[[98  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
>>> print(a2)  # New array
[98  2  3  4  5  6  7  8  9 10 11 12]
```

Read more about `flatten` at **`ndarray.flatten`** and `ravel` at **`ravel`**.

How to access the docstring for more information

This section covers `help()`, `?`, `??`

When it comes to the data science ecosystem, Python and NumPy are built with the user in mind. One of the best examples of this is the built-in access to documentation. Every object contains the reference to a string, which is known as the **docstring**. In most cases, this docstring contains a quick and concise summary of the object and how to use it. Python has a built-in `help()` function that can help you access this information. This means that nearly any time you need more information, you can use `help()` to quickly find the information that you need.

For example:

```
>>> help(max)
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

Because access to additional information is so useful, IPython uses the `?` character as a shorthand for accessing this documentation along with other relevant information. IPython is a command shell for interactive computing in multiple languages. You can find more information about IPython here.

For example:

```
In [0]: max?
max(iterable, *[, default=obj, key=func]) -> value
max(arg1, arg2, *args, *[, key=func]) -> value

With a single iterable argument, return its biggest item. The
default keyword-only argument specifies an object to return if
the provided iterable is empty.
With two or more arguments, return the largest argument.
Type:      builtin_function_or_method
```

You can even use this notation for object methods and objects themselves.

Let’s say you create this array:

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

Then you can obtain a lot of useful information (first details about `a` itself, followed by the docstring of `ndarray` of which `a` is an instance):

```
In [1]: a?
Type:            ndarray
String form:     [1 2 3 4 5 6]
Length:          6
File:            ~/anaconda3/lib/python3.7/site-packages/numpy/__init__.py
Docstring:       <no docstring>
Class docstring:
ndarray(shape, dtype=float, buffer=None, offset=0,
        strides=None, order=None)

An array object represents a multidimensional, homogeneous array
of fixed-size items.  An associated data-type object describes the
format of each element in the array (its byte-order, how many bytes it
occupies in memory, whether it is an integer, a floating point number,
or something else, etc.)

Arrays should be constructed using `array`, `zeros` or `empty` (refer
to the See Also section below).  The parameters given here refer to
a low-level method (`ndarray(...)`) for instantiating an array.

For more information, refer to the `numpy` module and examine the
methods and attributes of an array.

Parameters
----------
(for the __new__ method; see Notes below)

shape : tuple of ints
        Shape of created array.
...
```

This also works for functions and other objects that **you** create. Just remember to include a docstring with your function using a string literal (`""" """` or `''' '''` around your documentation).

For example, if you create this function:

```
>>> def double(a):
...   '''Return a * 2'''
...   return a * 2
```

You can obtain information about the function:

```
In [2]: double?
Signature: double(a)
Docstring: Return a * 2
File:      ~/Desktop/<ipython-input-23-b5adf20be596>
Type:      function
```

You can reach another level of information by reading the source code of the object you're interested in. Using a double question mark (`??`) allows you to access the source code.

For example:

```
In [3]: double??
Signature: double(a)
Source:
def double(a):
    '''Return a * 2'''
    return a * 2
File:      ~/Desktop/<ipython-input-23-b5adf20be596>
Type:      function
```

If the object in question is compiled in a language other than Python, using `??` will return the same information as `?`. You'll find this with a lot of built-in objects and types, for example:

```
In [4]: len?
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

and :

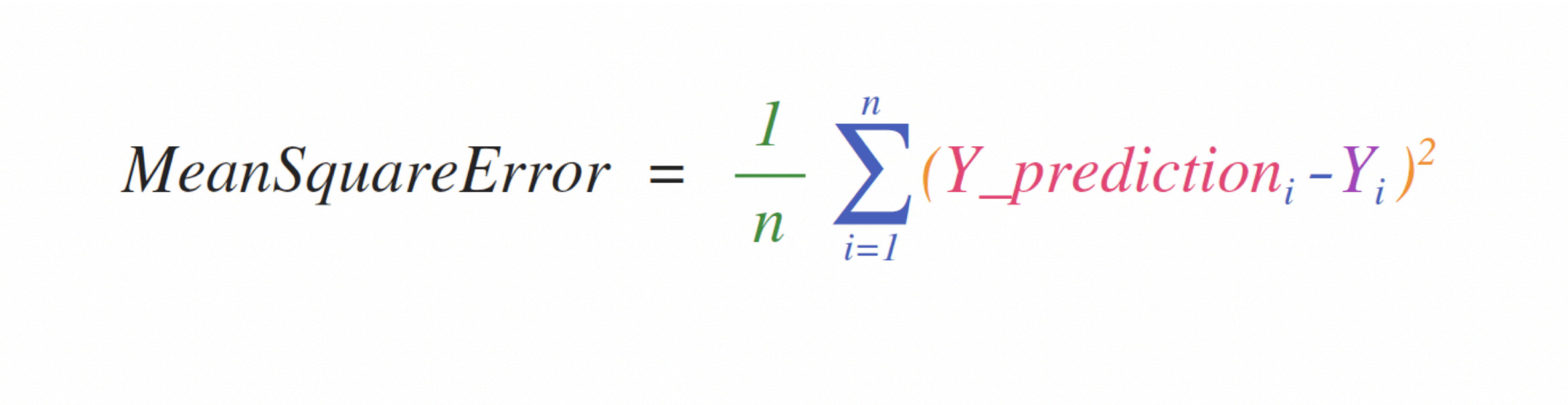
```
In [5]: len??
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

have the same output because they were compiled in a programming language other than Python.

Working with mathematical formulas

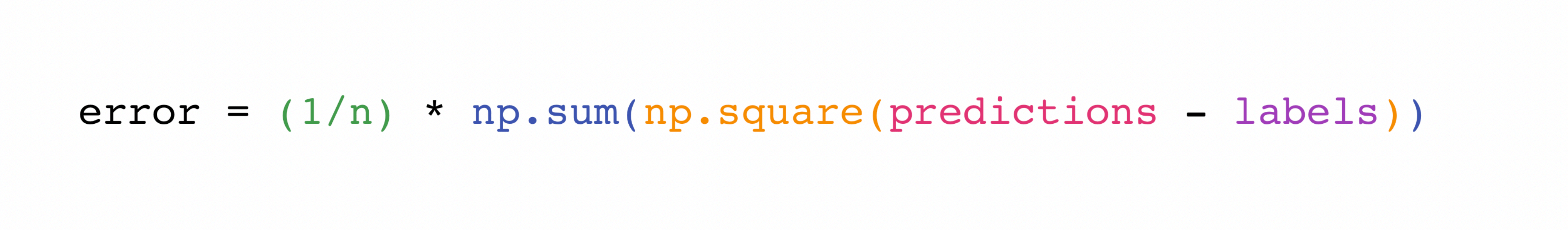
The ease of implementing mathematical formulas that work on arrays is one of the things that make NumPy so widely used in the scientific Python community.

For example, this is the mean square error formula (a central formula used in supervised machine learning models that deal with regression):



$$MeanSquareError = \frac{1}{n}\sum_{i=1}^{n}(Y\_prediction_i - Y_i)^2$$

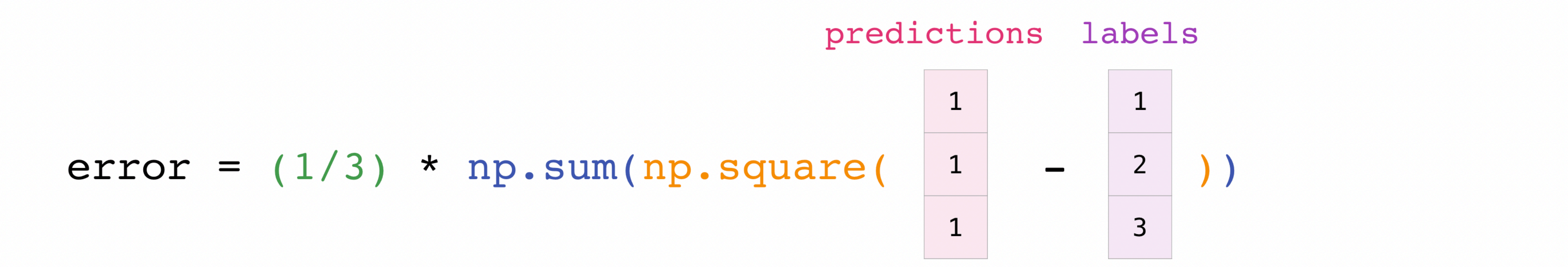
Implementing this formula is simple and straightforward in NumPy:



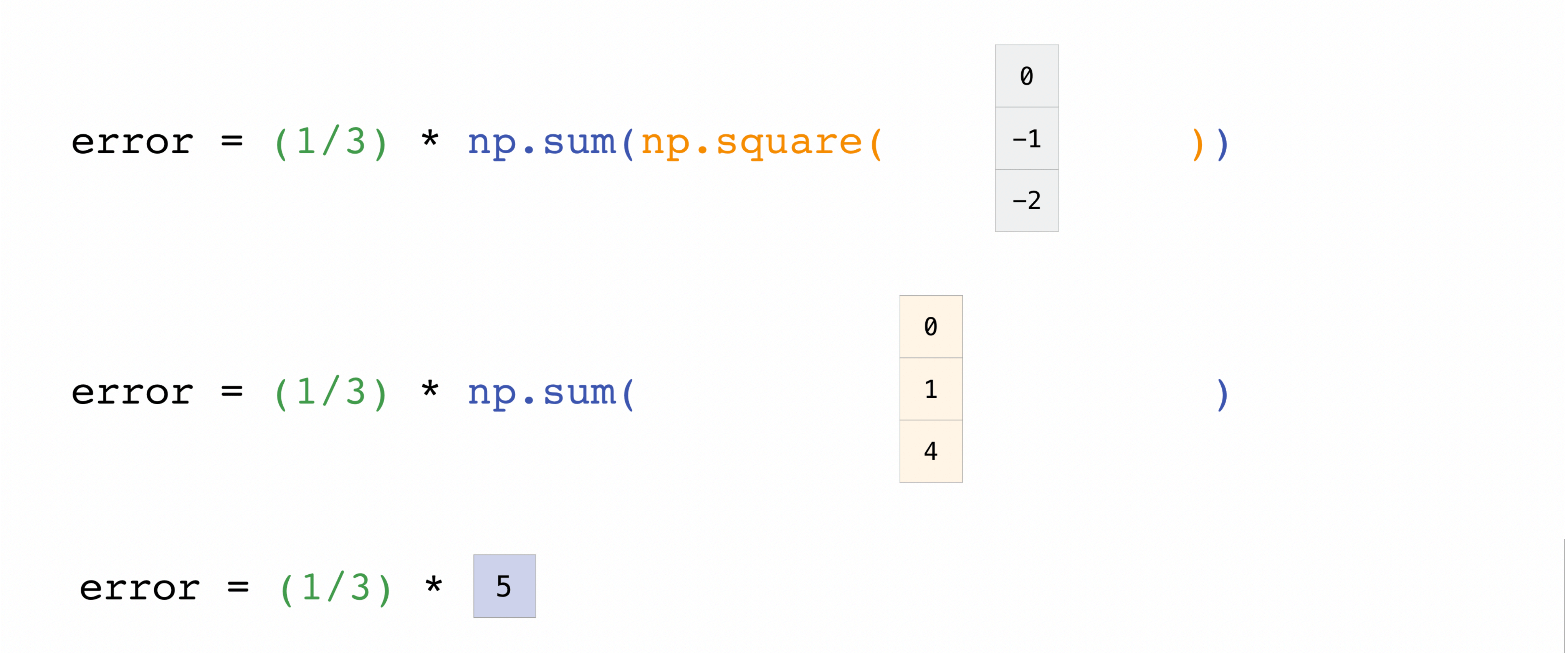
```
error = (1/n) * np.sum(np.square(predictions - labels))
```

What makes this work so well is that `predictions` and `labels` can contain one or a thousand values. They only need to be the same size.

You can visualize it this way:



In this example, both the predictions and labels vectors contain three values, meaning `n` has a value of three. After we carry out subtractions the values in the vector are squared. Then NumPy sums the values, and your result is the error value for that prediction and a score for the quality of the model.


error = (1/3) * np.sum(np.square(0 -1 -2))
error = (1/3) * np.sum(0 1 4)
error = (1/3) * 5

How to save and load NumPy objects

This section covers `np.save`, `np.savez`, `np.savetxt`, `np.load`, `np.loadtxt`

You will, at some point, want to save your arrays to disk and load them back without having to re-run the code. Fortunately, there are several ways to save and load objects with NumPy. The ndarray objects can be saved to and loaded from the disk files with `loadtxt` and `savetxt` functions that handle normal text files, `load` and `save` functions that handle NumPy binary files with a **.npy** file extension, and a `savez` function that handles NumPy files with a **.npz** file extension.

The **.npy** and **.npz** files store data, shape, dtype, and other information required to reconstruct the ndarray in a way that allows the array to be correctly retrieved, even when the file is on another machine with different architecture.

If you want to store a single ndarray object, store it as a .npy file using `np.save`. If you want to store more than one ndarray object in a single file, save it as a .npz file using `np.savez`. You can also save several arrays into a single file in compressed npz format with **`savez_compressed`**.

It's easy to save and load and array with `np.save()`. Just make sure to specify the array you want to save and a file name. For example, if you create this array:

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

You can save it as "filename.npy" with:

```
>>> np.save('filename', a)
```

You can use `np.load()` to reconstruct your array.

```
>>> b = np.load('filename.npy')
```

If you want to check your array, you can run::

```
>>> print(b)
[1 2 3 4 5 6]
```

You can save a NumPy array as a plain text file like a **.csv** or **.txt** file with `np.savetxt`.

For example, if you create this array:

```
>>> csv_arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
```

You can easily save it as a .csv file with the name “new_file.csv” like this:

```
>>> np.savetxt('new_file.csv', csv_arr)
```

You can quickly and easily load your saved text file using `loadtxt()`:

```
>>> np.loadtxt('new_file.csv')
array([1., 2., 3., 4., 5., 6., 7., 8.])
```

The `savetxt()` and `loadtxt()` functions accept additional optional parameters such as header, footer, and delimiter. While text files can be easier for sharing, .npy and .npz files are smaller and faster to read. If you need more sophisticated handling of your text file (for example, if you need to work with lines that contain missing values), you will want to use the **`genfromtxt`** function.

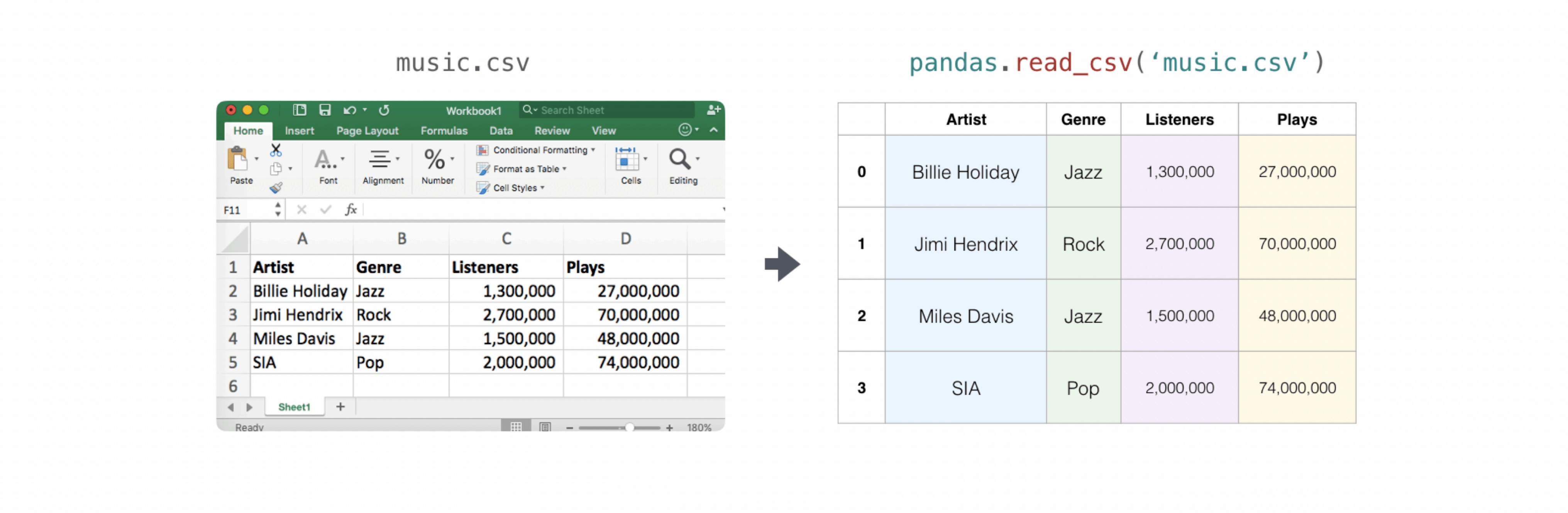
With **`savetxt`**, you can specify headers, footers, comments, and more.

Learn more about input and output routines here.

Importing and exporting a CSV

It’s simple to read in a CSV that contains existing information. The best and easiest way to do this is to use Pandas.

```
>>> import pandas as pd

>>> # If all of your columns are the same type:
>>> x = pd.read_csv('music.csv', header=0).values
>>> print(x)
[['Billie Holiday' 'Jazz' 1300000 27000000]
 ['Jimmie Hendrix' 'Rock' 2700000 70000000]
 ['Miles Davis' 'Jazz' 1500000 48000000]
 ['SIA' 'Pop' 2000000 74000000]]

>>> # You can also simply select the columns you need:
>>> x = pd.read_csv('music.csv', usecols=['Artist', 'Plays']).values
>>> print(x)
[['Billie Holiday' 27000000]
 ['Jimmie Hendrix' 70000000]
 ['Miles Davis' 48000000]
 ['SIA' 74000000]]
```



	A	B	C	D
1	Artist	Genre	Listeners	Plays
2	Billie Holiday	Jazz	1,300,000	27,000,000
3	Jimi Hendrix	Rock	2,700,000	70,000,000
4	Miles Davis	Jazz	1,500,000	48,000,000
5	SIA	Pop	2,000,000	74,000,000
6				

	Artist	Genre	Listeners	Plays
0	Billie Holiday	Jazz	1,300,000	27,000,000
1	Jimi Hendrix	Rock	2,700,000	70,000,000
2	Miles Davis	Jazz	1,500,000	48,000,000
3	SIA	Pop	2,000,000	74,000,000

It’s simple to use Pandas in order to export your array as well. If you are new to NumPy, you may want to create a Pandas dataframe from the values in your array and then write the data frame to a CSV file with Pandas.

If you created this array “a”

```
>>> a = np.array([[-2.58289208,  0.43014843, -1.24082018, 1.59572603],
...               [ 0.99027828, 1.17150989,  0.94125714, -0.14692469],
...               [ 0.76989341,  0.81299683, -0.95068423, 0.11769564],
...               [ 0.20484034,  0.34784527,  1.96979195, 0.51992837]])
```

You could create a Pandas dataframe

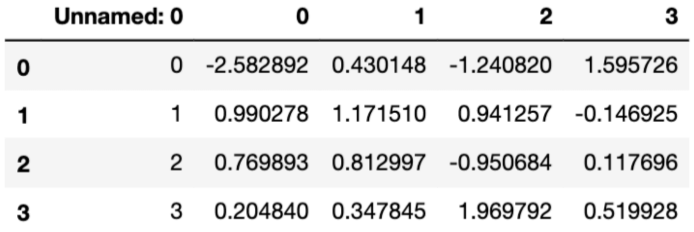
```
>>> df = pd.DataFrame(a)
>>> print(df)
          0         1         2         3
0 -2.582892  0.430148 -1.240820  1.595726
1  0.990278  1.171510  0.941257 -0.146925
2  0.769893  0.812997 -0.950684  0.117696
3  0.204840  0.347845  1.969792  0.519928
```

You can easily save your dataframe with:

```
>>> df.to_csv('pd.csv')
```

And read your CSV with:

```
>>> data = pd.read_csv('pd.csv')
```



	Unnamed: 0	0	1	2	3
0	0	-2.582892	0.430148	-1.240820	1.595726
1	1	0.990278	1.171510	0.941257	-0.146925
2	2	0.769893	0.812997	-0.950684	0.117696
3	3	0.204840	0.347845	1.969792	0.519928

You can also save your array with the NumPy `savetxt` method.

```
>>> np.savetxt('np.csv', a, fmt='%.2f', delimiter=',', header='1,  2,  3,  4')
```

If you're using the command line, you can read your saved CSV any time with a command such as:

```
$ cat np.csv
#  1,  2,  3,  4
-2.58,0.43,-1.24,1.60
0.99,1.17,0.94,-0.15
0.77,0.81,-0.95,0.12
0.20,0.35,1.97,0.52
```

Or you can open the file any time with a text editor!

If you're interested in learning more about Pandas, take a look at the official Pandas documentation. Learn how to install Pandas with the official Pandas installation information.

Plotting arrays with Matplotlib

If you need to generate a plot for your values, it's very simple with Matplotlib.

For example, you may have an array like this one:

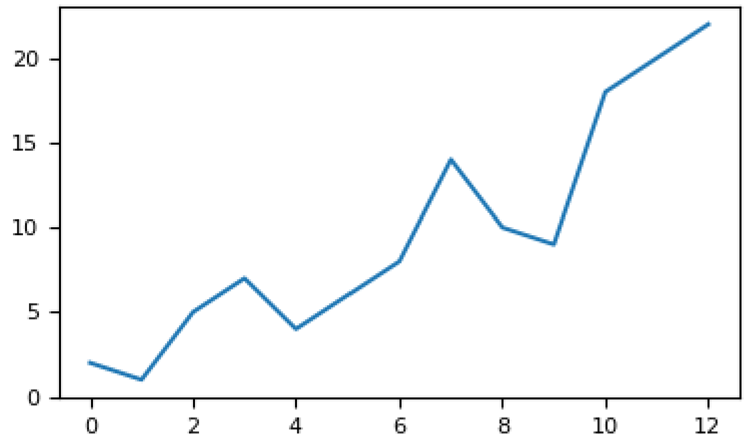
```
>>> a = np.array([2, 1, 5, 7, 4, 6, 8, 14, 10, 9, 18, 20, 22])
```

If you already have Matplotlib installed, you can import it with:

```
>>> import matplotlib.pyplot as plt

# If you're using Jupyter Notebook, you may also want to run the following
# line of code to display your code in the notebook:

%matplotlib inline
```

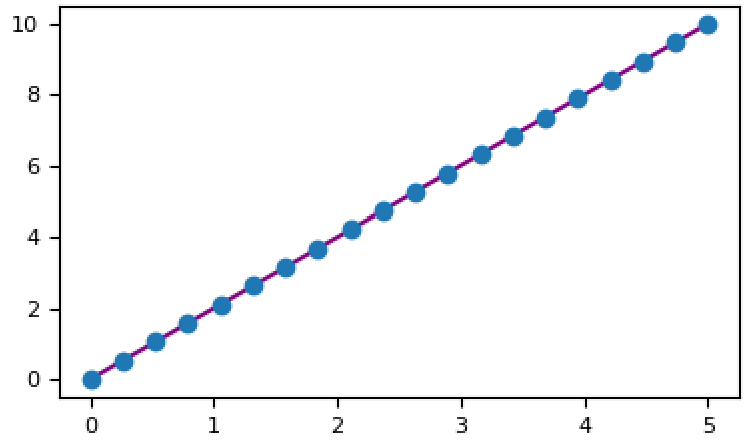
All you need to do to plot your values is run:

```
>>> plt.plot(a)

# If you are running from a command line, you may need to do this:
# >>> plt.show()
```



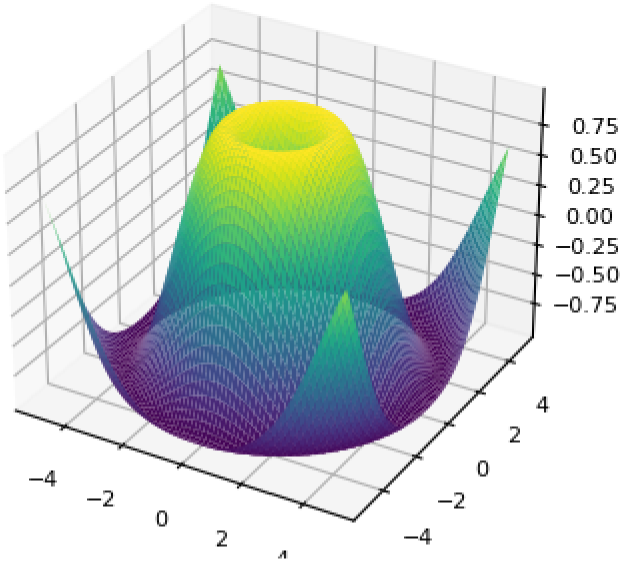
For example, you can plot a 1D array like this:

```
>>> x = np.linspace(0, 5, 20)
>>> y = np.linspace(0, 10, 20)
>>> plt.plot(x, y, 'purple') # line
>>> plt.plot(x, y, 'o')      # dots
```



With Matplotlib, you have access to an enormous number of visualization options.

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = Axes3D(fig)
>>> X = np.arange(-5, 5, 0.15)
>>> Y = np.arange(-5, 5, 0.15)
>>> X, Y = np.meshgrid(X, Y)
>>> R = np.sqrt(X**2 + Y**2)
>>> Z = np.sin(R)

>>> ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis')
```



To read more about Matplotlib and what it can do, take a look at the official documentation. For directions regarding installing Matplotlib, see the official installation section.

Image credits: Jay Alammar http://jalammar.github.io/

<< NumPy quickstart

NumPy basics >>

Last updated on Jan 31, 2021.
Created using Sphinx 2.4.4.

CHAPTER

TWO

USER GUIDE

The User Guide covers all of pandas by topic area. Each of the subsections introduces a topic (such as “working with missing data”), and discusses how pandas approaches the problem, with many examples throughout.

Users brand-new to pandas should start with 10min.

For a high level summary of the pandas fundamentals, see *Intro to data structures* and *Essential basic functionality*.

Further information on any specific method can be obtained in the *API reference*. {{ header }}

2.1 10 minutes to pandas

This is a short introduction to pandas, geared mainly for new users. You can see more complex recipes in the *Cookbook*.

Customarily, we import as follows:

```
In [1]: import numpy as np

In [2]: import pandas as pd
```

2.1.1 Object creation

See the *Data Structure Intro section*.

Creating a `Series` by passing a list of values, letting pandas create a default integer index:

```
In [3]: s = pd.Series([1, 3, 5, np.nan, 6, 8])

In [4]: s
Out[4]:
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Creating a `DataFrame` by passing a NumPy array, with a datetime index and labeled columns:

```
In [5]: dates = pd.date_range("20130101", periods=6)

In [6]: dates
```

(continues on next page)

(continued from previous page)

```
Out[6]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

In [7]: df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))

In [8]: df
Out[8]:
                   A         B         C         D
2013-01-01  1.712776  0.336509 -0.945427  0.449729
2013-01-02 -0.177498  1.487290 -0.458333 -0.721907
2013-01-03 -0.139201 -2.090971  0.975329 -0.565294
2013-01-04  0.248276 -1.001166  0.126691 -0.381556
2013-01-05  0.279357 -0.002987 -0.566766  1.929640
2013-01-06 -1.161636 -0.594994  0.722651  0.895291
```

Creating a `DataFrame` by passing a dict of objects that can be converted to series-like.

```
In [9]: df2 = pd.DataFrame(
   ...:     {
   ...:         "A": 1.0,
   ...:         "B": pd.Timestamp("20130102"),
   ...:         "C": pd.Series(1, index=list(range(4)), dtype="float32"),
   ...:         "D": np.array([3] * 4, dtype="int32"),
   ...:         "E": pd.Categorical(["test", "train", "test", "train"]),
   ...:         "F": "foo",
   ...:     }
   ...: )
   ...:

In [10]: df2
Out[10]:
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
```

The columns of the resulting `DataFrame` have different *dtypes*.

```
In [11]: df2.dtypes
Out[11]:
A          float64
B   datetime64[ns]
C          float32
D            int32
E         category
F           object
dtype: object
```

If you're using IPython, tab completion for column names (as well as public attributes) is automatically enabled. Here's a subset of the attributes that will be completed:

```
In [12]: df2.<TAB>  # noqa: E225, E999
df2.A                  df2.bool
```

(continues on next page)

(continued from previous page)

```
df2.abs                  df2.boxplot
df2.add                  df2.C
df2.add_prefix           df2.clip
df2.add_suffix           df2.columns
df2.align                df2.copy
df2.all                  df2.count
df2.any                  df2.combine
df2.append               df2.D
df2.apply                df2.describe
df2.applymap             df2.diff
df2.B                    df2.duplicated
```

As you can see, the columns `A`, `B`, `C`, and `D` are automatically tab completed. `E` and `F` are there as well; the rest of the attributes have been truncated for brevity.

2.1.2 Viewing data

See the *Basics section*.

Here is how to view the top and bottom rows of the frame:

```
In [13]: df.head()
Out[13]: 
                   A         B         C         D
2013-01-01  1.712776  0.336509 -0.945427  0.449729
2013-01-02 -0.177498  1.487290 -0.458333 -0.721907
2013-01-03 -0.139201 -2.090971  0.975329 -0.565294
2013-01-04  0.248276 -1.001166  0.126691 -0.381556
2013-01-05  0.279357 -0.002987 -0.566766  1.929640

In [14]: df.tail(3)
Out[14]: 
                   A         B         C         D
2013-01-04  0.248276 -1.001166  0.126691 -0.381556
2013-01-05  0.279357 -0.002987 -0.566766  1.929640
2013-01-06 -1.161636 -0.594994  0.722651  0.895291
```

Display the index, columns:

```
In [15]: df.index
Out[15]: 
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

In [16]: df.columns
Out[16]: Index(['A', 'B', 'C', 'D'], dtype='object')
```

`DataFrame.to_numpy()` gives a NumPy representation of the underlying data. Note that this can be an expensive operation when your `DataFrame` has columns with different data types, which comes down to a fundamental difference between pandas and NumPy: **NumPy arrays have one dtype for the entire array, while pandas DataFrames have one dtype per column**. When you call `DataFrame.to_numpy()`, pandas will find the NumPy dtype that can hold *all* of the dtypes in the DataFrame. This may end up being `object`, which requires casting every value to a Python object.

For df, our DataFrame of all floating-point values, DataFrame.to_numpy() is fast and doesn't require copying data.

```
In [17]: df.to_numpy()
Out[17]:
array([[ 1.71277599,  0.33650864, -0.94542653,  0.44972919],
       [-0.17749784,  1.48729025, -0.45833313, -0.72190726],
       [-0.1392012 , -2.09097127,  0.97532913, -0.5652943 ],
       [ 0.248276  , -1.00116557,  0.12669091, -0.38155642],
       [ 0.27935679, -0.00298719, -0.56676602,  1.92963962],
       [-1.16163559, -0.59499379,  0.72265057,  0.89529069]])
```

For df2, the DataFrame with multiple dtypes, DataFrame.to_numpy() is relatively expensive.

```
In [18]: df2.to_numpy()
Out[18]:
array([[1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo']],
      dtype=object)
```

Note: DataFrame.to_numpy() does *not* include the index or column labels in the output.

describe() shows a quick statistic summary of your data:

```
In [19]: df.describe()
Out[19]:
              A         B         C         D
count  6.000000  6.000000  6.000000  6.000000
mean   0.127012 -0.311053 -0.024309  0.267650
std    0.935604  1.222560  0.763036  1.027987
min   -1.161636 -2.090971 -0.945427 -0.721907
25%   -0.167924 -0.899623 -0.539658 -0.519360
50%    0.054537 -0.298990 -0.165821  0.034086
75%    0.271587  0.251635  0.573661  0.783900
max    1.712776  1.487290  0.975329  1.929640
```

Transposing your data:

```
In [20]: df.T
Out[20]:
   2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-06
A    1.712776   -0.177498   -0.139201    0.248276    0.279357   -1.161636
B    0.336509    1.487290   -2.090971   -1.001166   -0.002987   -0.594994
C   -0.945427   -0.458333    0.975329    0.126691   -0.566766    0.722651
D    0.449729   -0.721907   -0.565294   -0.381556    1.929640    0.895291
```

Sorting by an axis:

```
In [21]: df.sort_index(axis=1, ascending=False)
Out[21]:
                   D         C         B         A
2013-01-01  0.449729 -0.945427  0.336509  1.712776
2013-01-02 -0.721907 -0.458333  1.487290 -0.177498
2013-01-03 -0.565294  0.975329 -2.090971 -0.139201
```

(continues on next page)

(continued from previous page)

```
2013-01-04 -0.381556  0.126691 -1.001166  0.248276
2013-01-05  1.929640 -0.566766 -0.002987  0.279357
2013-01-06  0.895291  0.722651 -0.594994 -1.161636
```

Sorting by values:

```
In [22]: df.sort_values(by="B")
Out[22]:
                   A         B         C         D
2013-01-03 -0.139201 -2.090971  0.975329 -0.565294
2013-01-04  0.248276 -1.001166  0.126691 -0.381556
2013-01-06 -1.161636 -0.594994  0.722651  0.895291
2013-01-05  0.279357 -0.002987 -0.566766  1.929640
2013-01-01  1.712776  0.336509 -0.945427  0.449729
2013-01-02 -0.177498  1.487290 -0.458333 -0.721907
```

2.1.3 Selection

Note: While standard Python / NumPy expressions for selecting and setting are intuitive and come in handy for interactive work, for production code, we recommend the optimized pandas data access methods, `.at`, `.iat`, `.loc` and `.iloc`.

See the indexing documentation *Indexing and Selecting Data* and *MultiIndex / Advanced Indexing*.

Getting

Selecting a single column, which yields a `Series`, equivalent to `df.A`:

```
In [23]: df["A"]
Out[23]:
2013-01-01    1.712776
2013-01-02   -0.177498
2013-01-03   -0.139201
2013-01-04    0.248276
2013-01-05    0.279357
2013-01-06   -1.161636
Freq: D, Name: A, dtype: float64
```

Selecting via `[]`, which slices the rows.

```
In [24]: df[0:3]
Out[24]:
                   A         B         C         D
2013-01-01  1.712776  0.336509 -0.945427  0.449729
2013-01-02 -0.177498  1.487290 -0.458333 -0.721907
2013-01-03 -0.139201 -2.090971  0.975329 -0.565294

In [25]: df["20130102":"20130104"]
Out[25]:
                   A         B         C         D
2013-01-02 -0.177498  1.487290 -0.458333 -0.721907
```

(continues on next page)

(continued from previous page)

```
2013-01-03 -0.139201 -2.090971  0.975329 -0.565294
2013-01-04  0.248276 -1.001166  0.126691 -0.381556
```

Selection by label

See more in *Selection by Label*.

For getting a cross section using a label:

```
In [26]: df.loc[dates[0]]
Out[26]:
A    1.712776
B    0.336509
C   -0.945427
D    0.449729
Name: 2013-01-01 00:00:00, dtype: float64
```

Selecting on a multi-axis by label:

```
In [27]: df.loc[:, ["A", "B"]]
Out[27]:
                   A         B
2013-01-01  1.712776  0.336509
2013-01-02 -0.177498  1.487290
2013-01-03 -0.139201 -2.090971
2013-01-04  0.248276 -1.001166
2013-01-05  0.279357 -0.002987
2013-01-06 -1.161636 -0.594994
```

Showing label slicing, both endpoints are *included*:

```
In [28]: df.loc["20130102":"20130104", ["A", "B"]]
Out[28]:
                   A         B
2013-01-02 -0.177498  1.487290
2013-01-03 -0.139201 -2.090971
2013-01-04  0.248276 -1.001166
```

Reduction in the dimensions of the returned object:

```
In [29]: df.loc["20130102", ["A", "B"]]
Out[29]:
A   -0.177498
B    1.487290
Name: 2013-01-02 00:00:00, dtype: float64
```

For getting a scalar value:

```
In [30]: df.loc[dates[0], "A"]
Out[30]: 1.7127759912633918
```

For getting fast access to a scalar (equivalent to the prior method):

```
In [31]: df.at[dates[0], "A"]
Out[31]: 1.7127759912633918
```

Selection by position

See more in *Selection by Position*.

Select via the position of the passed integers:

```
In [32]: df.iloc[3]
Out[32]:
A    0.248276
B   -1.001166
C    0.126691
D   -0.381556
Name: 2013-01-04 00:00:00, dtype: float64
```

By integer slices, acting similar to NumPy/Python:

```
In [33]: df.iloc[3:5, 0:2]
Out[33]:
                   A         B
2013-01-04  0.248276 -1.001166
2013-01-05  0.279357 -0.002987
```

By lists of integer position locations, similar to the NumPy/Python style:

```
In [34]: df.iloc[[1, 2, 4], [0, 2]]
Out[34]:
                   A         C
2013-01-02 -0.177498 -0.458333
2013-01-03 -0.139201  0.975329
2013-01-05  0.279357 -0.566766
```

For slicing rows explicitly:

```
In [35]: df.iloc[1:3, :]
Out[35]:
                   A         B         C         D
2013-01-02 -0.177498  1.487290 -0.458333 -0.721907
2013-01-03 -0.139201 -2.090971  0.975329 -0.565294
```

For slicing columns explicitly:

```
In [36]: df.iloc[:, 1:3]
Out[36]:
                   B         C
2013-01-01  0.336509 -0.945427
2013-01-02  1.487290 -0.458333
2013-01-03 -2.090971  0.975329
2013-01-04 -1.001166  0.126691
2013-01-05 -0.002987 -0.566766
2013-01-06 -0.594994  0.722651
```

For getting a value explicitly:

```
In [37]: df.iloc[1, 1]
Out[37]: 1.48729024797565
```

For getting fast access to a scalar (equivalent to the prior method):

```
In [38]: df.iat[1, 1]
Out[38]: 1.48729024797565
```

Boolean indexing

Using a single column's values to select data.

```
In [39]: df[df["A"] > 0]
Out[39]: 
                   A         B         C         D
2013-01-01  1.712776  0.336509 -0.945427  0.449729
2013-01-04  0.248276 -1.001166  0.126691 -0.381556
2013-01-05  0.279357 -0.002987 -0.566766  1.929640
```

Selecting values from a DataFrame where a boolean condition is met.

```
In [40]: df[df > 0]
Out[40]: 
                   A         B         C         D
2013-01-01  1.712776  0.336509       NaN  0.449729
2013-01-02       NaN  1.487290       NaN       NaN
2013-01-03       NaN       NaN  0.975329       NaN
2013-01-04  0.248276       NaN  0.126691       NaN
2013-01-05  0.279357       NaN       NaN  1.929640
2013-01-06       NaN       NaN  0.722651  0.895291
```

Using the `isin()` method for filtering:

```
In [41]: df2 = df.copy()

In [42]: df2["E"] = ["one", "one", "two", "three", "four", "three"]

In [43]: df2
Out[43]: 
                   A         B         C         D      E
2013-01-01  1.712776  0.336509 -0.945427  0.449729    one
2013-01-02 -0.177498  1.487290 -0.458333 -0.721907    one
2013-01-03 -0.139201 -2.090971  0.975329 -0.565294    two
2013-01-04  0.248276 -1.001166  0.126691 -0.381556  three
2013-01-05  0.279357 -0.002987 -0.566766  1.929640   four
2013-01-06 -1.161636 -0.594994  0.722651  0.895291  three

In [44]: df2[df2["E"].isin(["two", "four"])]
Out[44]: 
                   A         B         C         D     E
2013-01-03 -0.139201 -2.090971  0.975329 -0.565294   two
2013-01-05  0.279357 -0.002987 -0.566766  1.929640  four
```

Setting

Setting a new column automatically aligns the data by the indexes.

```
In [45]: s1 = pd.Series([1, 2, 3, 4, 5, 6], index=pd.date_range("20130102",␣
↪periods=6))

In [46]: s1
Out[46]:
2013-01-02    1
2013-01-03    2
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64

In [47]: df["F"] = s1
```

Setting values by label:

```
In [48]: df.at[dates[0], "A"] = 0
```

Setting values by position:

```
In [49]: df.iat[0, 1] = 0
```

Setting by assigning with a NumPy array:

```
In [50]: df.loc[:, "D"] = np.array([5] * len(df))
```

The result of the prior setting operations.

```
In [51]: df
Out[51]:
                   A         B         C  D    F
2013-01-01  0.000000  0.000000 -0.945427  5  NaN
2013-01-02 -0.177498  1.487290 -0.458333  5  1.0
2013-01-03 -0.139201 -2.090971  0.975329  5  2.0
2013-01-04  0.248276 -1.001166  0.126691  5  3.0
2013-01-05  0.279357 -0.002987 -0.566766  5  4.0
2013-01-06 -1.161636 -0.594994  0.722651  5  5.0
```

A `where` operation with setting.

```
In [52]: df2 = df.copy()

In [53]: df2[df2 > 0] = -df2

In [54]: df2
Out[54]:
                   A         B         C  D    F
2013-01-01  0.000000  0.000000 -0.945427 -5  NaN
2013-01-02 -0.177498 -1.487290 -0.458333 -5 -1.0
2013-01-03 -0.139201 -2.090971 -0.975329 -5 -2.0
2013-01-04 -0.248276 -1.001166 -0.126691 -5 -3.0
2013-01-05 -0.279357 -0.002987 -0.566766 -5 -4.0
2013-01-06 -1.161636 -0.594994 -0.722651 -5 -5.0
```

2.1.4 Missing data

pandas primarily uses the value `np.nan` to represent missing data. It is by default not included in computations. See the *Missing Data section*.

Reindexing allows you to change/add/delete the index on a specified axis. This returns a copy of the data.

```
In [55]: df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ["E"])

In [56]: df1.loc[dates[0] : dates[1], "E"] = 1

In [57]: df1
Out[57]: 
                   A         B         C  D    F    E
2013-01-01  0.000000  0.000000 -0.945427  5  NaN  1.0
2013-01-02 -0.177498  1.487290 -0.458333  5  1.0  1.0
2013-01-03 -0.139201 -2.090971  0.975329  5  2.0  NaN
2013-01-04  0.248276 -1.001166  0.126691  5  3.0  NaN
```

To drop any rows that have missing data.

```
In [58]: df1.dropna(how="any")
Out[58]: 
                   A        B         C  D    F    E
2013-01-02 -0.177498  1.48729 -0.458333  5  1.0  1.0
```

Filling missing data.

```
In [59]: df1.fillna(value=5)
Out[59]: 
                   A         B         C  D    F    E
2013-01-01  0.000000  0.000000 -0.945427  5  5.0  1.0
2013-01-02 -0.177498  1.487290 -0.458333  5  1.0  1.0
2013-01-03 -0.139201 -2.090971  0.975329  5  2.0  5.0
2013-01-04  0.248276 -1.001166  0.126691  5  3.0  5.0
```

To get the boolean mask where values are `nan`.

```
In [60]: pd.isna(df1)
Out[60]: 
                A      B      C      D      F      E
2013-01-01  False  False  False  False   True  False
2013-01-02  False  False  False  False  False  False
2013-01-03  False  False  False  False  False   True
2013-01-04  False  False  False  False  False   True
```

2.1.5 Operations

See the *Basic section on Binary Ops*.

Stats

Operations in general *exclude* missing data.

Performing a descriptive statistic:

```
In [61]: df.mean()
Out[61]:
A   -0.158450
B   -0.367138
C   -0.024309
D    5.000000
F    3.000000
dtype: float64
```

Same operation on the other axis:

```
In [62]: df.mean(1)
Out[62]:
2013-01-01    1.013643
2013-01-02    1.370292
2013-01-03    1.149031
2013-01-04    1.474760
2013-01-05    1.741921
2013-01-06    1.793204
Freq: D, dtype: float64
```

Operating with objects that have different dimensionality and need alignment. In addition, pandas automatically broadcasts along the specified dimension.

```
In [63]: s = pd.Series([1, 3, 5, np.nan, 6, 8], index=dates).shift(2)

In [64]: s
Out[64]:
2013-01-01    NaN
2013-01-02    NaN
2013-01-03    1.0
2013-01-04    3.0
2013-01-05    5.0
2013-01-06    NaN
Freq: D, dtype: float64

In [65]: df.sub(s, axis="index")
Out[65]:
                   A         B         C    D    F
2013-01-01       NaN       NaN       NaN  NaN  NaN
2013-01-02       NaN       NaN       NaN  NaN  NaN
2013-01-03 -1.139201 -3.090971 -0.024671  4.0  1.0
2013-01-04 -2.751724 -4.001166 -2.873309  2.0  0.0
2013-01-05 -4.720643 -5.002987 -5.566766  0.0 -1.0
2013-01-06       NaN       NaN       NaN  NaN  NaN
```

Apply

Applying functions to the data:

```
In [66]: df.apply(np.cumsum)
Out[66]:
                   A         B         C   D     F
2013-01-01  0.000000  0.000000 -0.945427   5   NaN
2013-01-02 -0.177498  1.487290 -1.403760  10   1.0
2013-01-03 -0.316699 -0.603681 -0.428431  15   3.0
2013-01-04 -0.068423 -1.604847 -0.301740  20   6.0
2013-01-05  0.210934 -1.607834 -0.868506  25  10.0
2013-01-06 -0.950702 -2.202828 -0.145855  30  15.0

In [67]: df.apply(lambda x: x.max() - x.min())
Out[67]:
A    1.440992
B    3.578262
C    1.920756
D    0.000000
F    4.000000
dtype: float64
```

Histogramming

See more at *Histogramming and Discretization*.

```
In [68]: s = pd.Series(np.random.randint(0, 7, size=10))

In [69]: s
Out[69]:
0    1
1    2
2    0
3    5
4    1
5    0
6    0
7    3
8    5
9    0
dtype: int64

In [70]: s.value_counts()
Out[70]:
0    4
1    2
5    2
2    1
3    1
dtype: int64
```

String Methods

Series is equipped with a set of string processing methods in the `str` attribute that make it easy to operate on each element of the array, as in the code snippet below. Note that pattern-matching in `str` generally uses regular expressions by default (and in some cases always uses them). See more at *Vectorized String Methods*.

```
In [71]: s = pd.Series(["A", "B", "C", "Aaba", "Baca", np.nan, "CABA", "dog", "cat"])

In [72]: s.str.lower()
Out[72]:
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

2.1.6 Merge

Concat

pandas provides various facilities for easily combining together Series and DataFrame objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

See the *Merging section*.

Concatenating pandas objects together with `concat()`:

```
In [73]: df = pd.DataFrame(np.random.randn(10, 4))

In [74]: df
Out[74]:
          0         1         2         3
0  1.689662 -1.311349 -0.229433  0.687357
1 -1.225525 -1.555269  1.647132  2.163551
2  1.133407  0.362450  0.238007  0.166870
3 -0.050427  0.091661 -0.532976  0.071773
4  1.633323  1.360126 -0.509585  0.008281
5 -0.874520 -1.203754  0.542787 -0.488790
6  1.960843 -0.555579  0.762490  0.614574
7  0.580310 -0.200614  0.501313 -1.571428
8 -0.818894 -1.714748 -0.110789 -1.783606
9  0.257285 -0.300526  1.599902  0.247784

# break it into pieces
In [75]: pieces = [df[:3], df[3:7], df[7:]]

In [76]: pd.concat(pieces)
Out[76]:
          0         1         2         3
0  1.689662 -1.311349 -0.229433  0.687357
1 -1.225525 -1.555269  1.647132  2.163551
```

(continues on next page)

(continued from previous page)

```
2  1.133407  0.362450  0.238007  0.166870
3 -0.050427  0.091661 -0.532976  0.071773
4  1.633323  1.360126 -0.509585  0.008281
5 -0.874520 -1.203754  0.542787 -0.488790
6  1.960843 -0.555579  0.762490  0.614574
7  0.580310 -0.200614  0.501313 -1.571428
8 -0.818894 -1.714748 -0.110789 -1.783606
9  0.257285 -0.300526  1.599902  0.247784
```

Note: Adding a column to a `DataFrame` is relatively fast. However, adding a row requires a copy, and may be expensive. We recommend passing a pre-built list of records to the `DataFrame` constructor instead of building a `DataFrame` by iteratively appending records to it. See *Appending to dataframe* for more.

Join

SQL style merges. See the *Database style joining* section.

```
In [77]: left = pd.DataFrame({"key": ["foo", "foo"], "lval": [1, 2]})

In [78]: right = pd.DataFrame({"key": ["foo", "foo"], "rval": [4, 5]})

In [79]: left
Out[79]:
   key  lval
0  foo     1
1  foo     2

In [80]: right
Out[80]:
   key  rval
0  foo     4
1  foo     5

In [81]: pd.merge(left, right, on="key")
Out[81]:
   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
```

Another example that can be given is:

```
In [82]: left = pd.DataFrame({"key": ["foo", "bar"], "lval": [1, 2]})

In [83]: right = pd.DataFrame({"key": ["foo", "bar"], "rval": [4, 5]})

In [84]: left
Out[84]:
   key  lval
0  foo     1
1  bar     2
```

(continues on next page)

(continued from previous page)

```
In [85]: right
Out[85]:
   key  rval
0  foo     4
1  bar     5

In [86]: pd.merge(left, right, on="key")
Out[86]:
   key  lval  rval
0  foo     1     4
1  bar     2     5
```

2.1.7 Grouping

By “group by” we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria
- **Applying** a function to each group independently
- **Combining** the results into a data structure

See the *Grouping section*.

```
In [87]: df = pd.DataFrame(
   ....:     {
   ....:         "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
   ....:         "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
   ....:         "C": np.random.randn(8),
   ....:         "D": np.random.randn(8),
   ....:     }
   ....: )
   ....:

In [88]: df
Out[88]:
     A      B         C         D
0  foo    one  1.355302 -0.578010
1  bar    one  1.433322  0.238339
2  foo    two  0.158418  1.066454
3  bar  three  0.196638  0.327690
4  foo    two -1.244050  2.601324
5  bar    two  0.986477 -0.845308
6  foo    one -0.770289  0.183200
7  foo  three  0.443799  0.418662
```

Grouping and then applying the `sum()` function to the resulting groups.

```
In [89]: df.groupby("A").sum()
Out[89]:
            C         D
A
bar  2.616436 -0.279278
foo -0.056820  3.691630
```

Grouping by multiple columns forms a hierarchical index, and again we can apply the `sum()` function.

```
In [90]: df.groupby(["A", "B"]).sum()
Out[90]:
                  C         D
A   B
bar one    1.433322  0.238339
    three  0.196638  0.327690
    two    0.986477 -0.845308
foo one    0.585014 -0.394810
    three  0.443799  0.418662
    two   -1.085632  3.667778
```

2.1.8 Reshaping

See the sections on *Hierarchical Indexing* and *Reshaping*.

Stack

```
In [91]: tuples = list(
   ....:     zip(
   ....:         *[
   ....:             ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
   ....:             ["one", "two", "one", "two", "one", "two", "one", "two"],
   ....:         ]
   ....:     )
   ....: )
   ....:

In [92]: index = pd.MultiIndex.from_tuples(tuples, names=["first", "second"])

In [93]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=["A", "B"])

In [94]: df2 = df[:4]

In [95]: df2
Out[95]:
                     A         B
first second
bar   one    -0.908885  0.134150
      two    -0.322778 -0.084722
baz   one    -0.095210 -2.256921
      two     2.615210  0.214028
```

The `stack()` method “compresses” a level in the DataFrame’s columns.

```
In [96]: stacked = df2.stack()

In [97]: stacked
Out[97]:
first  second
bar    one     A   -0.908885
               B    0.134150
       two     A   -0.322778
               B   -0.084722
baz    one     A   -0.095210
```

(continues on next page)

(continued from previous page)

```
               B   -2.256921
       two     A    2.615210
               B    0.214028
dtype: float64
```

With a "stacked" DataFrame or Series (having a `MultiIndex` as the `index`), the inverse operation of `stack()` is `unstack()`, which by default unstacks the **last level**:

```
In [98]: stacked.unstack()
Out[98]: 
                     A         B
first second                    
bar   one    -0.908885  0.134150
      two    -0.322778 -0.084722
baz   one    -0.095210 -2.256921
      two     2.615210  0.214028

In [99]: stacked.unstack(1)
Out[99]: 
second        one       two
first                      
bar   A -0.908885 -0.322778
      B  0.134150 -0.084722
baz   A -0.095210  2.615210
      B -2.256921  0.214028

In [100]: stacked.unstack(0)
Out[100]: 
first          bar       baz
second                      
one    A -0.908885 -0.095210
       B  0.134150 -2.256921
two    A -0.322778  2.615210
       B -0.084722  0.214028
```

Pivot tables

See the section on *Pivot Tables*.

```
In [101]: df = pd.DataFrame(
   .....:     {
   .....:         "A": ["one", "one", "two", "three"] * 3,
   .....:         "B": ["A", "B", "C"] * 4,
   .....:         "C": ["foo", "foo", "foo", "bar", "bar", "bar"] * 2,
   .....:         "D": np.random.randn(12),
   .....:         "E": np.random.randn(12),
   .....:     }
   .....: )
   .....: 

In [102]: df
Out[102]: 
        A  B    C         D         E
0     one  A  foo  0.745325 -1.492997
1     one  B  foo -1.297547 -0.339886
```

(continues on next page)

(continued from previous page)

```
2     two  C  foo -0.964388  0.482778
3   three  A  bar  1.079703  1.604137
4     one  B  bar  0.675036  0.560282
5     one  C  bar  0.408994  0.558150
6     two  A  foo  0.261266 -0.149187
7   three  B  foo  1.012788 -1.226392
8     one  C  foo -0.105359  1.401395
9     one  A  bar  1.180264 -0.625340
10    two  B  bar  1.648181 -1.047040
11  three  C  bar  0.792630 -1.237315
```

We can produce pivot tables from this data very easily:

```
In [103]: pd.pivot_table(df, values="D", index=["A", "B"], columns=["C"])
Out[103]:
C             bar       foo
A     B
one   A  1.180264  0.745325
      B  0.675036 -1.297547
      C  0.408994 -0.105359
three A  1.079703       NaN
      B       NaN  1.012788
      C  0.792630       NaN
two   A       NaN  0.261266
      B  1.648181       NaN
      C       NaN -0.964388
```

2.1.9 Time series

pandas has simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data). This is extremely common in, but not limited to, financial applications. See the *Time Series section*.

```
In [104]: rng = pd.date_range("1/1/2012", periods=100, freq="S")

In [105]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

In [106]: ts.resample("5Min").sum()
Out[106]:
2012-01-01    26891
Freq: 5T, dtype: int64
```

Time zone representation:

```
In [107]: rng = pd.date_range("3/6/2012 00:00", periods=5, freq="D")

In [108]: ts = pd.Series(np.random.randn(len(rng)), rng)

In [109]: ts
Out[109]:
2012-03-06    0.174529
2012-03-07    0.457967
2012-03-08   -0.314052
2012-03-09   -0.072789
2012-03-10   -1.091168
```

(continues on next page)

(continued from previous page)

```
Freq: D, dtype: float64

In [110]: ts_utc = ts.tz_localize("UTC")

In [111]: ts_utc
Out[111]:
2012-03-06 00:00:00+00:00    0.174529
2012-03-07 00:00:00+00:00    0.457967
2012-03-08 00:00:00+00:00   -0.314052
2012-03-09 00:00:00+00:00   -0.072789
2012-03-10 00:00:00+00:00   -1.091168
Freq: D, dtype: float64
```

Converting to another time zone:

```
In [112]: ts_utc.tz_convert("US/Eastern")
Out[112]:
2012-03-05 19:00:00-05:00    0.174529
2012-03-06 19:00:00-05:00    0.457967
2012-03-07 19:00:00-05:00   -0.314052
2012-03-08 19:00:00-05:00   -0.072789
2012-03-09 19:00:00-05:00   -1.091168
Freq: D, dtype: float64
```

Converting between time span representations:

```
In [113]: rng = pd.date_range("1/1/2012", periods=5, freq="M")

In [114]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [115]: ts
Out[115]:
2012-01-31    1.262722
2012-02-29    0.203150
2012-03-31    0.129981
2012-04-30    0.376551
2012-05-31    0.103878
Freq: M, dtype: float64

In [116]: ps = ts.to_period()

In [117]: ps
Out[117]:
2012-01    1.262722
2012-02    0.203150
2012-03    0.129981
2012-04    0.376551
2012-05    0.103878
Freq: M, dtype: float64

In [118]: ps.to_timestamp()
Out[118]:
2012-01-01    1.262722
2012-02-01    0.203150
2012-03-01    0.129981
2012-04-01    0.376551
2012-05-01    0.103878
```

(continues on next page)

(continued from previous page)

```
Freq: MS, dtype: float64
```

Converting between period and timestamp enables some convenient arithmetic functions to be used. In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end:

```
In [119]: prng = pd.period_range("1990Q1", "2000Q4", freq="Q-NOV")

In [120]: ts = pd.Series(np.random.randn(len(prng)), prng)

In [121]: ts.index = (prng.asfreq("M", "e") + 1).asfreq("H", "s") + 9

In [122]: ts.head()
Out[122]: 
1990-03-01 09:00    0.124480
1990-06-01 09:00    0.406012
1990-09-01 09:00    0.203973
1990-12-01 09:00    0.355695
1991-03-01 09:00    2.962668
Freq: H, dtype: float64
```

2.1.10 Categoricals

pandas can include categorical data in a DataFrame. For full docs, see the *categorical introduction* and the *API documentation*.

```
In [123]: df = pd.DataFrame(
   .....:     {"id": [1, 2, 3, 4, 5, 6], "raw_grade": ["a", "b", "b", "a", "a", "e"]}
   .....: )
   .....: 
```

Convert the raw grades to a categorical data type.

```
In [124]: df["grade"] = df["raw_grade"].astype("category")

In [125]: df["grade"]
Out[125]: 
0    a
1    b
2    b
3    a
4    a
5    e
Name: grade, dtype: category
Categories (3, object): ['a', 'b', 'e']
```

Rename the categories to more meaningful names (assigning to Series.cat.categories() is in place!).

```
In [126]: df["grade"].cat.categories = ["very good", "good", "very bad"]
```

Reorder the categories and simultaneously add the missing categories (methods under Series.cat() return a new Series by default).

```
In [127]: df["grade"] = df["grade"].cat.set_categories(
   .....:     ["very bad", "bad", "medium", "good", "very good"]
   .....: )
   .....:

In [128]: df["grade"]
Out[128]:
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (5, object): ['very bad', 'bad', 'medium', 'good', 'very good']
```

Sorting is per order in the categories, not lexical order.

```
In [129]: df.sort_values(by="grade")
Out[129]:
   id raw_grade      grade
5   6         e   very bad
1   2         b       good
2   3         b       good
0   1         a  very good
3   4         a  very good
4   5         a  very good
```

Grouping by a categorical column also shows empty categories.

```
In [130]: df.groupby("grade").size()
Out[130]:
grade
very bad     1
bad          0
medium       0
good         2
very good    3
dtype: int64
```

2.1.11 Plotting

See the *Plotting* docs.

We use the standard convention for referencing the matplotlib API:

```
In [131]: import matplotlib.pyplot as plt

In [132]: plt.close("all")
```

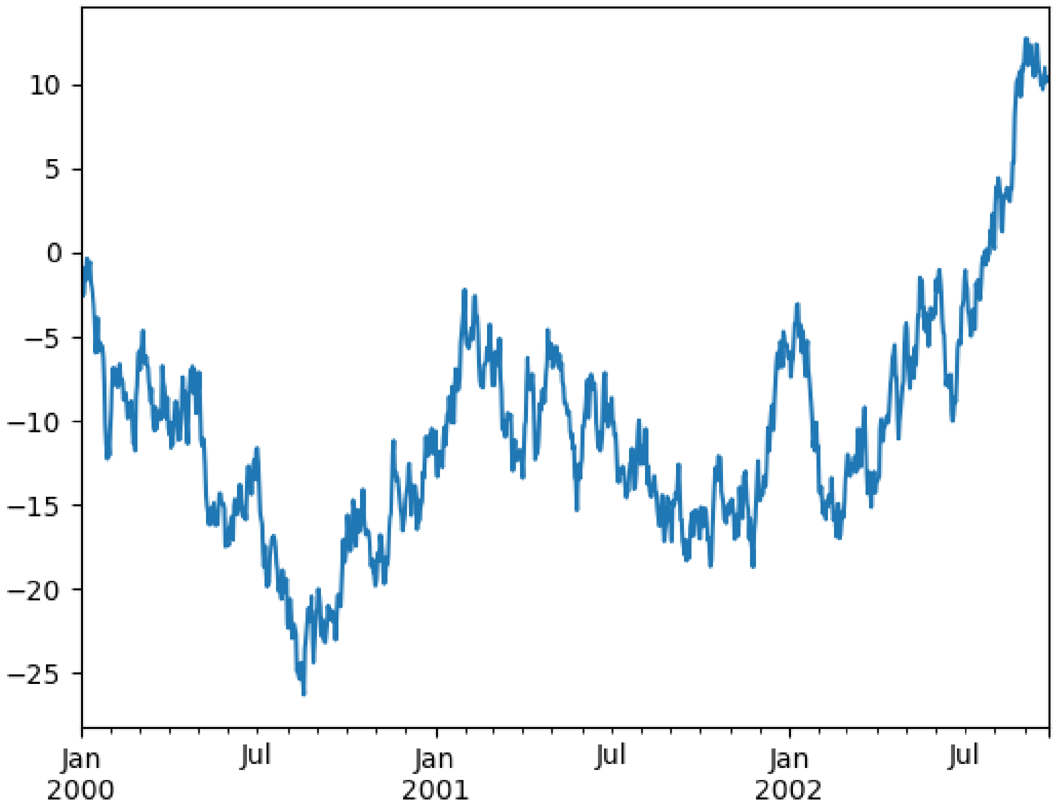
The `close()` method is used to close a figure window.

```
In [133]: ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000",␣
↪periods=1000))

In [134]: ts = ts.cumsum()
```

(continues on next page)

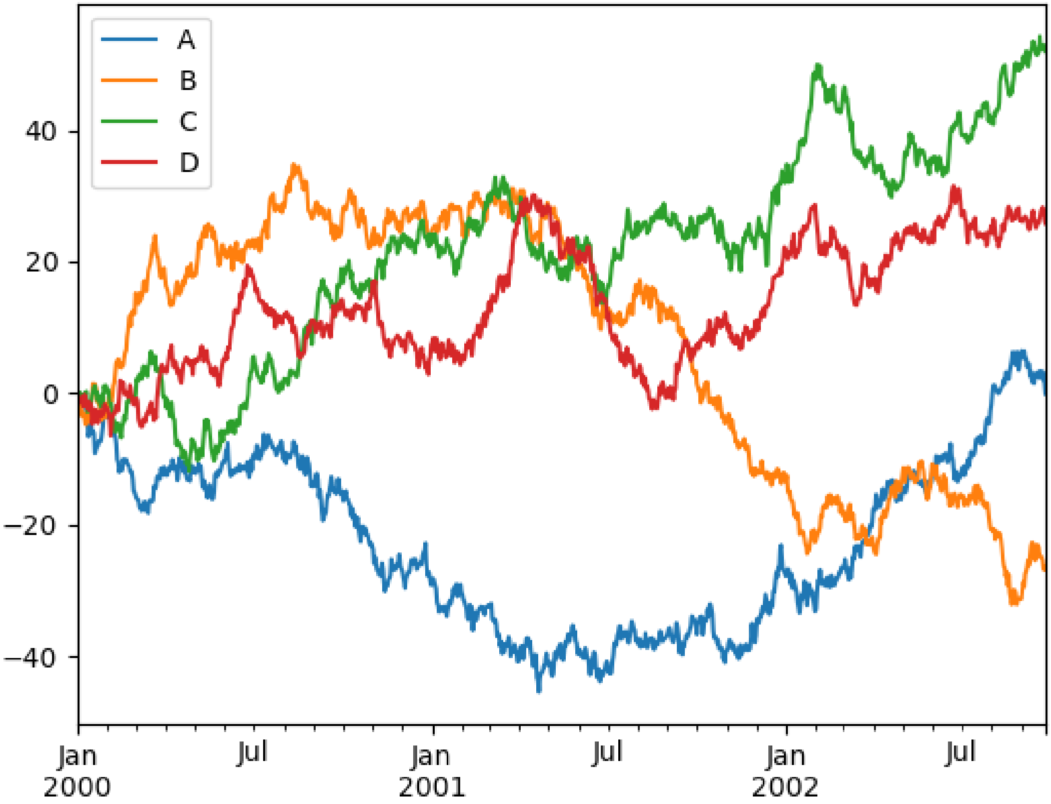
(continued from previous page)

```
In [135]: ts.plot();
```



On a DataFrame, the `plot()` method is a convenience to plot all of the columns with labels:

```
In [136]: df = pd.DataFrame(
   .....:     np.random.randn(1000, 4), index=ts.index, columns=["A", "B", "C", "D"]
   .....: )
   .....:

In [137]: df = df.cumsum()

In [138]: plt.figure();

In [139]: df.plot();

In [140]: plt.legend(loc='best');
```



2.1.12 Getting data in/out

CSV

Writing to a csv file.

```
In [141]: df.to_csv("foo.csv")
```

Reading from a csv file.

```
In [142]: pd.read_csv("foo.csv")
Out[142]:
     Unnamed: 0         A          B          C          D
0    2000-01-01 -1.065194  -0.002634  -0.760958  -0.031015
1    2000-01-02 -1.215840  -0.589713   0.060526  -1.667720
2    2000-01-03 -1.018093  -0.473960  -0.930696  -0.735057
3    2000-01-04 -0.574559  -2.144053  -0.009284  -1.372000
4    2000-01-05 -1.215664  -3.426502  -0.283470  -0.781581
..          ...       ...        ...        ...        ...
995  2002-09-22  2.281268 -24.942834  52.881026  27.864387
996  2002-09-23  3.271586 -25.498553  52.312461  28.249057
997  2002-09-24  1.913191 -26.822220  52.965859  27.642595
998  2002-09-25  1.690795 -26.325976  52.159921  26.247726
```

(continues on next page)

(continued from previous page)

```
999  2002-09-26 -0.098981 -26.778316  52.012625  25.801426

[1000 rows x 5 columns]
```

HDF5

Reading and writing to *HDFStores*.

Writing to a HDF5 Store.

```
In [143]: df.to_hdf("foo.h5", "df")
```

Reading from a HDF5 Store.

```
In [144]: pd.read_hdf("foo.h5", "df")
Out[144]:
                   A          B          C          D
2000-01-01 -1.065194  -0.002634  -0.760958  -0.031015
2000-01-02 -1.215840  -0.589713   0.060526  -1.667720
2000-01-03 -1.018093  -0.473960  -0.930696  -0.735057
2000-01-04 -0.574559  -2.144053  -0.009284  -1.372000
2000-01-05 -1.215664  -3.426502  -0.283470  -0.781581
...              ...        ...        ...        ...
2002-09-22  2.281268 -24.942834  52.881026  27.864387
2002-09-23  3.271586 -25.498553  52.312461  28.249057
2002-09-24  1.913191 -26.822220  52.965859  27.642595
2002-09-25  1.690795 -26.325976  52.159921  26.247726
2002-09-26 -0.098981 -26.778316  52.012625  25.801426

[1000 rows x 4 columns]
```

Excel

Reading and writing to *MS Excel*.

Writing to an excel file.

```
In [145]: df.to_excel("foo.xlsx", sheet_name="Sheet1")
```

Reading from an excel file.

```
In [146]: pd.read_excel("foo.xlsx", "Sheet1", index_col=None, na_values=["NA"])
Out[146]:
    Unnamed: 0         A          B          C          D
0   2000-01-01 -1.065194  -0.002634  -0.760958  -0.031015
1   2000-01-02 -1.215840  -0.589713   0.060526  -1.667720
2   2000-01-03 -1.018093  -0.473960  -0.930696  -0.735057
3   2000-01-04 -0.574559  -2.144053  -0.009284  -1.372000
4   2000-01-05 -1.215664  -3.426502  -0.283470  -0.781581
..         ...       ...        ...        ...        ...
995 2002-09-22  2.281268 -24.942834  52.881026  27.864387
996 2002-09-23  3.271586 -25.498553  52.312461  28.249057
997 2002-09-24  1.913191 -26.822220  52.965859  27.642595
998 2002-09-25  1.690795 -26.325976  52.159921  26.247726
```

(continues on next page)

(continued from previous page)

```
999 2002-09-26 -0.098981 -26.778316  52.012625  25.801426

[1000 rows x 5 columns]
```

2.1.13 Gotchas

If you are attempting to perform an operation you might see an exception like:

```
>>> if pd.Series([False, True, False]):
...     print("I was true")
Traceback
    ...
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

See *Comparisons* for an explanation and what to do.

See *Gotchas* as well.

2.2 Intro to data structures

We'll start with a quick, non-comprehensive overview of the fundamental data structures in pandas to get you started. The fundamental behavior about data types, indexing, and axis labeling / alignment apply across all of the objects. To get started, import NumPy and load pandas into your namespace:

```
In [1]: import numpy as np

In [2]: import pandas as pd
```

Here is a basic tenet to keep in mind: **data alignment is intrinsic**. The link between labels and data will not be broken unless done so explicitly by you.

We'll give a brief intro to the data structures, then consider all of the broad categories of functionality and methods in separate sections.

2.2.1 Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the **index**. The basic method to create a Series is to call:

```
>>> s = pd.Series(data, index=index)
```

Here, data can be many different things:

- a Python dict
- an ndarray
- a scalar value (like 5)

The passed **index** is a list of axis labels. Thus, this separates into a few cases depending on what **data is**:

From ndarray

If `data` is an ndarray, **index** must be the same length as **data**. If no index is passed, one will be created having values `[0, ..., len(data) - 1]`.

```
In [3]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])

In [4]: s
Out[4]: 
a    0.469112
b   -0.282863
c   -1.509059
d   -1.135632
e    1.212112
dtype: float64

In [5]: s.index
Out[5]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [6]: pd.Series(np.random.randn(5))
Out[6]: 
0   -0.173215
1    0.119209
2   -1.044236
3   -0.861849
4   -2.104569
dtype: float64
```

Note: pandas supports non-unique index values. If an operation that does not support duplicate index values is attempted, an exception will be raised at that time. The reason for being lazy is nearly all performance-based (there are many instances in computations, like parts of GroupBy, where the index is not used).

From dict

Series can be instantiated from dicts:

```
In [7]: d = {"b": 1, "a": 0, "c": 2}

In [8]: pd.Series(d)
Out[8]: 
b    1
a    0
c    2
dtype: int64
```

Note: When the data is a dict, and an index is not passed, the `Series` index will be ordered by the dict's insertion order, if you're using Python version >= 3.6 and pandas version >= 0.23.

If you're using Python < 3.6 or pandas < 0.23, and an index is not passed, the `Series` index will be the lexically ordered list of dict keys.

In the example above, if you were on a Python version lower than 3.6 or a pandas version lower than 0.23, the `Series` would be ordered by the lexical order of the dict keys (i.e. `['a', 'b', 'c']` rather than `['b', 'a', 'c']`).

If an index is passed, the values in data corresponding to the labels in the index will be pulled out.

```
In [9]: d = {"a": 0.0, "b": 1.0, "c": 2.0}

In [10]: pd.Series(d)
Out[10]:
a    0.0
b    1.0
c    2.0
dtype: float64

In [11]: pd.Series(d, index=["b", "c", "d", "a"])
Out[11]:
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Note: NaN (not a number) is the standard missing data marker used in pandas.

From scalar value

If `data` is a scalar value, an index must be provided. The value will be repeated to match the length of **index**.

```
In [12]: pd.Series(5.0, index=["a", "b", "c", "d", "e"])
Out[12]:
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Series is ndarray-like

`Series` acts very similarly to a `ndarray`, and is a valid argument to most NumPy functions. However, operations such as slicing will also slice the index.

```
In [13]: s[0]
Out[13]: 0.4691122999071863

In [14]: s[:3]
Out[14]:
a    0.469112
b   -0.282863
c   -1.509059
dtype: float64

In [15]: s[s > s.median()]
Out[15]:
a    0.469112
e    1.212112
dtype: float64

In [16]: s[[4, 3, 1]]
```

(continues on next page)

(continued from previous page)

```
Out[16]:
e    1.212112
d   -1.135632
b   -0.282863
dtype: float64

In [17]: np.exp(s)
Out[17]:
a    1.598575
b    0.753623
c    0.221118
d    0.321219
e    3.360575
dtype: float64
```

Note: We will address array-based indexing like `s[[4, 3, 1]]` in *section on indexing*.

Like a NumPy array, a pandas Series has a `dtype`.

```
In [18]: s.dtype
Out[18]: dtype('float64')
```

This is often a NumPy dtype. However, pandas and 3rd-party libraries extend NumPy's type system in a few places, in which case the dtype would be an `ExtensionDtype`. Some examples within pandas are *Categorical data* and *Nullable integer data type*. See *dtypes* for more.

If you need the actual array backing a `Series`, use `Series.array`.

```
In [19]: s.array
Out[19]:
<PandasArray>
[ 0.4691122999071863, -0.2828633443286633, -1.5090585031735124,
 -1.1356323710171934,  1.2121120250208506]
Length: 5, dtype: float64
```

Accessing the array can be useful when you need to do some operation without the index (to disable *automatic alignment*, for example).

`Series.array` will always be an `ExtensionArray`. Briefly, an ExtensionArray is a thin wrapper around one or more *concrete* arrays like a `numpy.ndarray`. pandas knows how to take an `ExtensionArray` and store it in a `Series` or a column of a `DataFrame`. See *dtypes* for more.

While Series is ndarray-like, if you need an *actual* ndarray, then use `Series.to_numpy()`.

```
In [20]: s.to_numpy()
Out[20]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])
```

Even if the Series is backed by a `ExtensionArray`, `Series.to_numpy()` will return a NumPy ndarray.

Series is dict-like

A Series is like a fixed-size dict in that you can get and set values by index label:

```
In [21]: s["a"]
Out[21]: 0.4691122999071863

In [22]: s["e"] = 12.0

In [23]: s
Out[23]:
a     0.469112
b    -0.282863
c    -1.509059
d    -1.135632
e    12.000000
dtype: float64

In [24]: "e" in s
Out[24]: True

In [25]: "f" in s
Out[25]: False
```

If a label is not contained, an exception is raised:

```
>>> s["f"]
KeyError: 'f'
```

Using the `get` method, a missing label will return None or specified default:

```
In [26]: s.get("f")

In [27]: s.get("f", np.nan)
Out[27]: nan
```

See also the *section on attribute access*.

Vectorized operations and label alignment with Series

When working with raw NumPy arrays, looping through value-by-value is usually not necessary. The same is true when working with Series in pandas. Series can also be passed into most NumPy methods expecting an ndarray.

```
In [28]: s + s
Out[28]:
a     0.938225
b    -0.565727
c    -3.018117
d    -2.271265
e    24.000000
dtype: float64

In [29]: s * 2
Out[29]:
a     0.938225
b    -0.565727
```

(continues on next page)

(continued from previous page)

```
c    -3.018117
d    -2.271265
e    24.000000
dtype: float64

In [30]: np.exp(s)
Out[30]:
a         1.598575
b         0.753623
c         0.221118
d         0.321219
e    162754.791419
dtype: float64
```

A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

```
In [31]: s[1:] + s[:-1]
Out[31]:
a         NaN
b   -0.565727
c   -3.018117
d   -2.271265
e         NaN
dtype: float64
```

The result of an operation between unaligned Series will have the **union** of the indexes involved. If a label is not found in one Series or the other, the result will be marked as missing `NaN`. Being able to write code without doing any explicit data alignment grants immense freedom and flexibility in interactive data analysis and research. The integrated data alignment features of the pandas data structures set pandas apart from the majority of related tools for working with labeled data.

Note: In general, we chose to make the default result of operations between differently indexed objects yield the **union** of the indexes in order to avoid loss of information. Having an index label, though the data is missing, is typically important information as part of a computation. You of course have the option of dropping labels with missing data via the **dropna** function.

Name attribute

Series can also have a `name` attribute:

```
In [32]: s = pd.Series(np.random.randn(5), name="something")

In [33]: s
Out[33]:
0   -0.494929
1    1.071804
2    0.721555
3   -0.706771
4   -1.039575
Name: something, dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [34]: s.name
Out[34]: 'something'
```

The Series `name` will be assigned automatically in many cases, in particular when taking 1D slices of DataFrame as you will see below.

You can rename a Series with the *`pandas.Series.rename()`* method.

```
In [35]: s2 = s.rename("different")

In [36]: s2.name
Out[36]: 'different'
```

Note that `s` and `s2` refer to different objects.

2.2.2 DataFrame

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A `Series`
- Another `DataFrame`

Along with the data, you can optionally pass **index** (row labels) and **columns** (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

If axis labels are not passed, they will be constructed from the input data based on common sense rules.

Note: When the data is a dict, and `columns` is not specified, the `DataFrame` columns will be ordered by the dict's insertion order, if you are using Python version >= 3.6 and pandas >= 0.23.

If you are using Python < 3.6 or pandas < 0.23, and `columns` is not specified, the `DataFrame` columns will be the lexically ordered list of dict keys.

From dict of Series or dicts

The resulting **index** will be the **union** of the indexes of the various Series. If there are any nested dicts, these will first be converted to Series. If no columns are passed, the columns will be the ordered list of dict keys.

```
In [37]: d = {
   ....:     "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]),
   ....:     "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c", "d"]),
   ....: }
   ....:

In [38]: df = pd.DataFrame(d)
```

(continues on next page)

(continued from previous page)

```
In [39]: df
Out[39]:
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

In [40]: pd.DataFrame(d, index=["d", "b", "a"])
Out[40]:
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0

In [41]: pd.DataFrame(d, index=["d", "b", "a"], columns=["two", "three"])
Out[41]:
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

The row and column labels can be accessed respectively by accessing the **index** and **columns** attributes:

Note: When a particular set of columns is passed along with a dict of data, the passed columns override the keys in the dict.

```
In [42]: df.index
Out[42]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [43]: df.columns
Out[43]: Index(['one', 'two'], dtype='object')
```

From dict of ndarrays / lists

The ndarrays must all be the same length. If an index is passed, it must clearly also be the same length as the arrays. If no index is passed, the result will be `range(n)`, where `n` is the array length.

```
In [44]: d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0, 1.0]}

In [45]: pd.DataFrame(d)
Out[45]:
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0

In [46]: pd.DataFrame(d, index=["a", "b", "c", "d"])
Out[46]:
   one  two
a  1.0  4.0
```

(continues on next page)

(continued from previous page)

```
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

From structured or record array

This case is handled identically to a dict of arrays.

```
In [47]: data = np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")])

In [48]: data[:] = [(1, 2.0, "Hello"), (2, 3.0, "World")]

In [49]: pd.DataFrame(data)
Out[49]: 
   A    B         C
0  1  2.0  b'Hello'
1  2  3.0  b'World'

In [50]: pd.DataFrame(data, index=["first", "second"])
Out[50]: 
        A    B         C
first   1  2.0  b'Hello'
second  2  3.0  b'World'

In [51]: pd.DataFrame(data, columns=["C", "A", "B"])
Out[51]: 
          C  A    B
0  b'Hello'  1  2.0
1  b'World'  2  3.0
```

Note: DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.

From a list of dicts

```
In [52]: data2 = [{"a": 1, "b": 2}, {"a": 5, "b": 10, "c": 20}]

In [53]: pd.DataFrame(data2)
Out[53]: 
   a   b     c
0  1   2   NaN
1  5  10  20.0

In [54]: pd.DataFrame(data2, index=["first", "second"])
Out[54]: 
        a   b     c
first   1   2   NaN
second  5  10  20.0

In [55]: pd.DataFrame(data2, columns=["a", "b"])
Out[55]: 
   a   b
```

(continues on next page)

(continued from previous page)

```
0  1   2
1  5  10
```

From a dict of tuples

You can automatically create a MultiIndexed frame by passing a tuples dictionary.

```
In [56]: pd.DataFrame(
   ....:     {
   ....:         ("a", "b"): {("A", "B"): 1, ("A", "C"): 2},
   ....:         ("a", "a"): {("A", "C"): 3, ("A", "B"): 4},
   ....:         ("a", "c"): {("A", "B"): 5, ("A", "C"): 6},
   ....:         ("b", "a"): {("A", "C"): 7, ("A", "B"): 8},
   ....:         ("b", "b"): {("A", "D"): 9, ("A", "B"): 10},
   ....:     }
   ....: )
   ....:
Out[56]:
       a              b
       b    a    c    a     b
A B  1.0  4.0  5.0  8.0  10.0
  C  2.0  3.0  6.0  7.0   NaN
  D  NaN  NaN  NaN  NaN   9.0
```

From a Series

The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).

From a list of namedtuples

The field names of the first `namedtuple` in the list determine the columns of the `DataFrame`. The remaining namedtuples (or tuples) are simply unpacked and their values are fed into the rows of the `DataFrame`. If any of those tuples is shorter than the first `namedtuple` then the later columns in the corresponding row are marked as missing values. If any are longer than the first `namedtuple`, a `ValueError` is raised.

```
In [57]: from collections import namedtuple

In [58]: Point = namedtuple("Point", "x y")

In [59]: pd.DataFrame([Point(0, 0), Point(0, 3), (2, 3)])
Out[59]:
   x  y
0  0  0
1  0  3
2  2  3

In [60]: Point3D = namedtuple("Point3D", "x y z")

In [61]: pd.DataFrame([Point3D(0, 0, 0), Point3D(0, 3, 5), Point(2, 3)])
Out[61]:
   x  y    z
```

(continues on next page)

(continued from previous page)

```
0  0  0  0.0
1  0  3  5.0
2  2  3  NaN
```

From a list of dataclasses

New in version 1.1.0.

Data Classes as introduced in PEP557, can be passed into the DataFrame constructor. Passing a list of dataclasses is equivalent to passing a list of dictionaries.

Please be aware, that all values in the list should be dataclasses, mixing types in the list would result in a TypeError.

```
In [62]: from dataclasses import make_dataclass

In [63]: Point = make_dataclass("Point", [("x", int), ("y", int)])

In [64]: pd.DataFrame([Point(0, 0), Point(0, 3), Point(2, 3)])
Out[64]: 
   x  y
0  0  0
1  0  3
2  2  3
```

Missing data

Much more will be said on this topic in the *Missing data* section. To construct a DataFrame with missing data, we use `np.nan` to represent missing values. Alternatively, you may pass a `numpy.MaskedArray` as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

Alternate constructors

DataFrame.from_dict

`DataFrame.from_dict` takes a dict of dicts or a dict of array-like sequences and returns a DataFrame. It operates like the `DataFrame` constructor except for the `orient` parameter which is `'columns'` by default, but which can be set to `'index'` in order to use the dict keys as row labels.

```
In [65]: pd.DataFrame.from_dict(dict([("A", [1, 2, 3]), ("B", [4, 5, 6])]))
Out[65]: 
   A  B
0  1  4
1  2  5
2  3  6
```

If you pass `orient='index'`, the keys will be the row labels. In this case, you can also pass the desired column names:

```
In [66]: pd.DataFrame.from_dict(
   ....:     dict([("A", [1, 2, 3]), ("B", [4, 5, 6])]),
   ....:     orient="index",
   ....:     columns=["one", "two", "three"],
   ....: )
   ....: 
Out[66]: 
```

(continues on next page)

(continued from previous page)

```
   one  two  three
A    1    2      3
B    4    5      6
```

DataFrame.from_records

DataFrame.from_records takes a list of tuples or an ndarray with structured dtype. It works analogously to the normal DataFrame constructor, except that the resulting DataFrame index may be a specific field of the structured dtype. For example:

```
In [67]: data
Out[67]: 
array([(1, 2., b'Hello'), (2, 3., b'World')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])

In [68]: pd.DataFrame.from_records(data, index="C")
Out[68]: 
          A    B
C               
b'Hello'  1  2.0
b'World'  2  3.0
```

Column selection, addition, deletion

You can treat a DataFrame semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations:

```
In [69]: df["one"]
Out[69]: 
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64

In [70]: df["three"] = df["one"] * df["two"]

In [71]: df["flag"] = df["one"] > 2

In [72]: df
Out[72]: 
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

Columns can be deleted or popped like with a dict:

```
In [73]: del df["two"]

In [74]: three = df.pop("three")

In [75]: df
Out[75]: 
```

(continues on next page)

(continued from previous page)

```
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
```

When inserting a scalar value, it will naturally be propagated to fill the column:

```
In [76]: df["foo"] = "bar"

In [77]: df
Out[77]:
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
c  3.0   True  bar
d  NaN  False  bar
```

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame's index:

```
In [78]: df["one_trunc"] = df["one"][:2]

In [79]: df
Out[79]:
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

You can insert raw ndarrays but their length must match the length of the DataFrame's index.

By default, columns get inserted at the end. The `insert` function is available to insert at a particular location in the columns:

```
In [80]: df.insert(1, "bar", df["one"])

In [81]: df
Out[81]:
   one  bar   flag  foo  one_trunc
a  1.0  1.0  False  bar        1.0
b  2.0  2.0  False  bar        2.0
c  3.0  3.0   True  bar        NaN
d  NaN  NaN  False  bar        NaN
```

Assigning new columns in method chains

Inspired by dplyr's `mutate` verb, DataFrame has an *`assign()`* method that allows you to easily create new columns that are potentially derived from existing columns.

```
In [82]: iris = pd.read_csv("data/iris.data")

In [83]: iris.head()
Out[83]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa

In [84]: iris.assign(sepal_ratio=iris["SepalWidth"] / iris["SepalLength"]).head()
Out[84]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

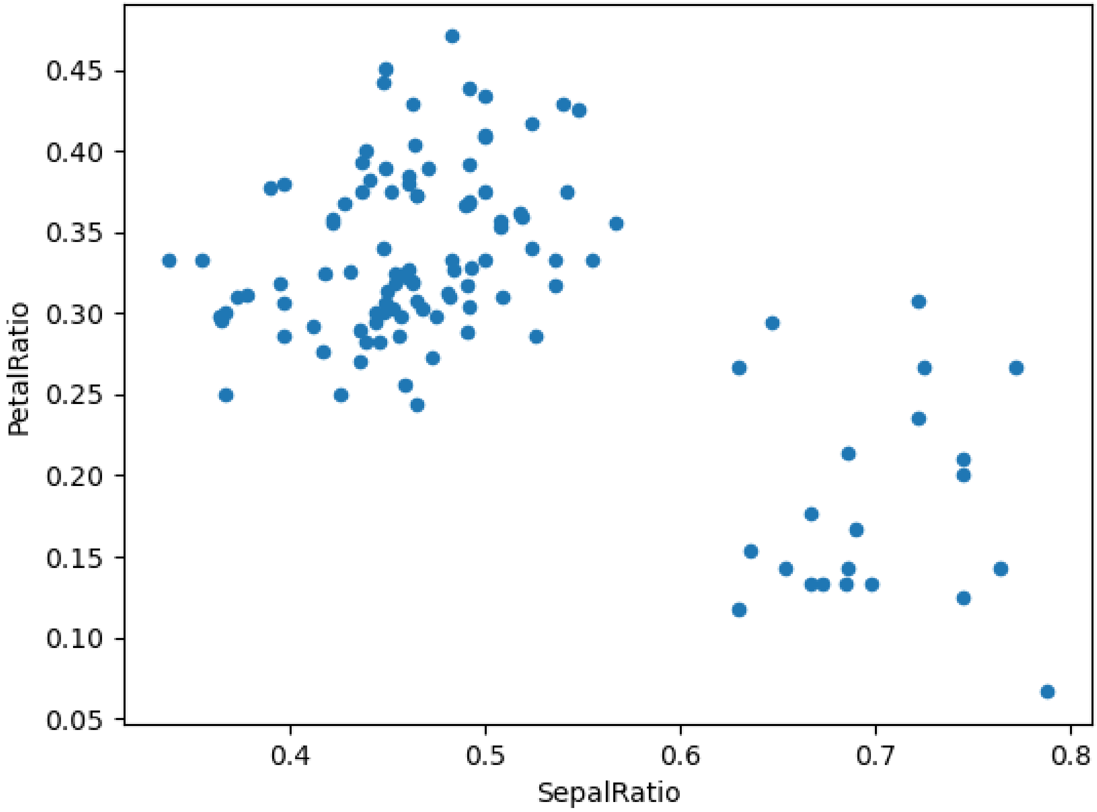
In the example above, we inserted a precomputed value. We can also pass in a function of one argument to be evaluated on the DataFrame being assigned to.

```
In [85]: iris.assign(sepal_ratio=lambda x: (x["SepalWidth"] / x["SepalLength"])).
↪head()
Out[85]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

`assign` **always** returns a copy of the data, leaving the original DataFrame untouched.

Passing a callable, as opposed to an actual value to be inserted, is useful when you don't have a reference to the DataFrame at hand. This is common when using `assign` in a chain of operations. For example, we can limit the DataFrame to just those observations with a Sepal Length greater than 5, calculate the ratio, and plot:

```
In [86]: (
   ....:     iris.query("SepalLength > 5")
   ....:     .assign(
   ....:         SepalRatio=lambda x: x.SepalWidth / x.SepalLength,
   ....:         PetalRatio=lambda x: x.PetalWidth / x.PetalLength,
   ....:     )
   ....:     .plot(kind="scatter", x="SepalRatio", y="PetalRatio")
   ....: )
   ....:
Out[86]: <AxesSubplot:xlabel='SepalRatio', ylabel='PetalRatio'>
```



Since a function is passed in, the function is computed on the DataFrame being assigned to. Importantly, this is the DataFrame that's been filtered to those rows with sepal length greater than 5. The filtering happens first, and then the ratio calculations. This is an example where we didn't have a reference to the *filtered* DataFrame available.

The function signature for `assign` is simply `**kwargs`. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a `Series` or NumPy array), or a function of one argument to be called on the `DataFrame`. A *copy* of the original DataFrame is returned, with the new values inserted.

Starting with Python 3.6 the order of `**kwargs` is preserved. This allows for *dependent* assignment, where an expression later in `**kwargs` can refer to a column created earlier in the same `assign()`.

```
In [87]: dfa = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})

In [88]: dfa.assign(C=lambda x: x["A"] + x["B"], D=lambda x: x["A"] + x["C"])
Out[88]:
   A  B  C   D
0  1  4  5   6
1  2  5  7   9
2  3  6  9  12
```

In the second expression, `x['C']` will refer to the newly created column, that's equal to `dfa['A'] + dfa['B']`.

Indexing / selection

The basics of indexing are as follows:

Operation	Syntax	Result
Select column	`df[col]`	Series
Select row by label	`df.loc[label]`	Series
Select row by integer location	`df.iloc[loc]`	Series
Slice rows	`df[5:10]`	DataFrame
Select rows by boolean vector	`df[bool_vec]`	DataFrame

Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```
In [89]: df.loc["b"]
Out[89]:
one              2.0
bar              2.0
flag           False
foo              bar
one_trunc        2.0
Name: b, dtype: object

In [90]: df.iloc[2]
Out[90]:
one             3.0
bar             3.0
flag           True
foo             bar
one_trunc       NaN
Name: c, dtype: object
```

For a more exhaustive treatment of sophisticated label-based indexing and slicing, see the *section on indexing*. We will address the fundamentals of reindexing / conforming to new sets of labels in the *section on reindexing*.

Data alignment and arithmetic

Data alignment between DataFrame objects automatically align on **both the columns and the index (row labels)**. Again, the resulting object will have the union of the column and row labels.

```
In [91]: df = pd.DataFrame(np.random.randn(10, 4), columns=["A", "B", "C", "D"])

In [92]: df2 = pd.DataFrame(np.random.randn(7, 3), columns=["A", "B", "C"])

In [93]: df + df2
Out[93]:
          A         B         C   D
0  0.045691 -0.014138  1.380871 NaN
1 -0.955398 -1.501007  0.037181 NaN
2 -0.662690  1.534833 -0.859691 NaN
3 -2.452949  1.237274 -0.133712 NaN
4  1.414490  1.951676 -2.320422 NaN
5 -0.494922 -1.649727 -1.084601 NaN
6 -1.047551 -0.748572 -0.805479 NaN
7       NaN       NaN       NaN NaN
```

(continues on next page)

(continued from previous page)

```
8       NaN       NaN       NaN NaN
9       NaN       NaN       NaN NaN
```

When doing an operation between DataFrame and Series, the default behavior is to align the Series **index** on the DataFrame **columns**, thus broadcasting row-wise. For example:

```
In [94]: df - df.iloc[0]
Out[94]: 
          A         B         C         D
0  0.000000  0.000000  0.000000  0.000000
1 -1.359261 -0.248717 -0.453372 -1.754659
2  0.253128  0.829678  0.010026 -1.991234
3 -1.311128  0.054325 -1.724913 -1.620544
4  0.573025  1.500742 -0.676070  1.367331
5 -1.741248  0.781993 -1.241620 -2.053136
6 -1.240774 -0.869551 -0.153282  0.000430
7 -0.743894  0.411013 -0.929563 -0.282386
8 -1.194921  1.320690  0.238224 -1.482644
9  2.293786  1.856228  0.773289 -1.446531
```

For explicit control over the matching and broadcasting behavior, see the section on *flexible binary operations*.

Operations with scalars are just as you would expect:

```
In [95]: df * 5 + 2
Out[95]: 
           A         B         C          D
0   3.359299 -0.124862  4.835102   3.381160
1  -3.437003 -1.368449  2.568242  -5.392133
2   4.624938  4.023526  4.885230  -6.575010
3  -3.196342  0.146766 -3.789461  -4.721559
4   6.224426  7.378849  1.454750  10.217815
5  -5.346940  3.785103 -1.373001  -6.884519
6  -2.844569 -4.472618  4.068691   3.383309
7  -0.360173  1.930201  0.187285   1.969232
8  -2.615303  6.478587  6.026220  -4.032059
9  14.828230  9.156280  8.701544  -3.851494

In [96]: 1 / df
Out[96]: 
          A          B         C           D
0  3.678365  -2.353094  1.763605    3.620145
1 -0.919624  -1.484363  8.799067   -0.676395
2  1.904807   2.470934  1.732964   -0.583090
3 -0.962215  -2.697986 -0.863638   -0.743875
4  1.183593   0.929567 -9.170108    0.608434
5 -0.680555   2.800959 -1.482360   -0.562777
6 -1.032084  -0.772485  2.416988    3.614523
7 -2.118489 -71.634509 -2.758294 -162.507295
8 -1.083352   1.116424  1.241860   -0.828904
9  0.389765   0.698687  0.746097   -0.854483

In [97]: df ** 4
Out[97]: 
           A             B         C             D
0   0.005462  3.261689e-02  0.103370  5.822320e-03
1   1.398165  2.059869e-01  0.000167  4.777482e+00
```

(continues on next page)

(continued from previous page)

```
2   0.075962  2.682596e-02  0.110877  8.650845e+00
3   1.166571  1.887302e-02  1.797515  3.265879e+00
4   0.509555  1.339298e+00  0.000141  7.297019e+00
5   4.661717  1.624699e-02  0.207103  9.969092e+00
6   0.881334  2.808277e+00  0.029302  5.858632e-03
7   0.049647  3.797614e-08  0.017276  1.433866e-09
8   0.725974  6.437005e-01  0.420446  2.118275e+00
9  43.329821  4.196326e+00  3.227153  1.875802e+00
```

Boolean operators work as well:

```
In [98]: df1 = pd.DataFrame({"a": [1, 0, 1], "b": [0, 1, 1]}, dtype=bool)

In [99]: df2 = pd.DataFrame({"a": [0, 1, 1], "b": [1, 1, 0]}, dtype=bool)

In [100]: df1 & df2
Out[100]: 
       a      b
0  False  False
1  False   True
2   True  False

In [101]: df1 | df2
Out[101]: 
      a     b
0  True  True
1  True  True
2  True  True

In [102]: df1 ^ df2
Out[102]: 
       a      b
0   True   True
1   True  False
2  False   True

In [103]: -df1
Out[103]: 
       a      b
0  False   True
1   True  False
2  False  False
```

Transposing

To transpose, access the `T` attribute (also the `transpose` function), similar to an ndarray:

```
# only show the first 5 rows
In [104]: df[:5].T
Out[104]: 
          0         1         2         3         4
A  0.271860 -1.087401  0.524988 -1.039268  0.844885
B -0.424972 -0.673690  0.404705 -0.370647  1.075770
C  0.567020  0.113648  0.577046 -1.157892 -0.109050
D  0.276232 -1.478427 -1.715002 -1.344312  1.643563
```

DataFrame interoperability with NumPy functions

Elementwise NumPy ufuncs (log, exp, sqrt, ...) and various other NumPy functions can be used with no issues on Series and DataFrame, assuming the data within are numeric:

```
In [105]: np.exp(df)
Out[105]:
           A         B         C         D
0   1.312403  0.653788  1.763006  1.318154
1   0.337092  0.509824  1.120358  0.227996
2   1.690438  1.498861  1.780770  0.179963
3   0.353713  0.690288  0.314148  0.260719
4   2.327710  2.932249  0.896686  5.173571
5   0.230066  1.429065  0.509360  0.169161
6   0.379495  0.274028  1.512461  1.318720
7   0.623732  0.986137  0.695904  0.993865
8   0.397301  2.449092  2.237242  0.299269
9  13.009059  4.183951  3.820223  0.310274

In [106]: np.asarray(df)
Out[106]:
array([[ 0.2719, -0.425 ,  0.567 ,  0.2762],
       [-1.0874, -0.6737,  0.1136, -1.4784],
       [ 0.525 ,  0.4047,  0.577 , -1.715 ],
       [-1.0393, -0.3706, -1.1579, -1.3443],
       [ 0.8449,  1.0758, -0.109 ,  1.6436],
       [-1.4694,  0.357 , -0.6746, -1.7769],
       [-0.9689, -1.2945,  0.4137,  0.2767],
       [-0.472 , -0.014 , -0.3625, -0.0062],
       [-0.9231,  0.8957,  0.8052, -1.2064],
       [ 2.5656,  1.4313,  1.3403, -1.1703]])
```

DataFrame is not intended to be a drop-in replacement for ndarray as its indexing semantics and data model are quite different in places from an n-dimensional array.

`Series` implements `__array_ufunc__`, which allows it to work with NumPy's universal functions.

The ufunc is applied to the underlying array in a Series.

```
In [107]: ser = pd.Series([1, 2, 3, 4])

In [108]: np.exp(ser)
Out[108]:
0     2.718282
1     7.389056
2    20.085537
3    54.598150
dtype: float64
```

Changed in version 0.25.0: When multiple `Series` are passed to a ufunc, they are aligned before performing the operation.

Like other parts of the library, pandas will automatically align labeled inputs as part of a ufunc with multiple inputs. For example, using `numpy.remainder()` on two `Series` with differently ordered labels will align before the operation.

```
In [109]: ser1 = pd.Series([1, 2, 3], index=["a", "b", "c"])

```

(continues on next page)

(continued from previous page)

```
In [110]: ser2 = pd.Series([1, 3, 5], index=["b", "a", "c"])

In [111]: ser1
Out[111]:
a    1
b    2
c    3
dtype: int64

In [112]: ser2
Out[112]:
b    1
a    3
c    5
dtype: int64

In [113]: np.remainder(ser1, ser2)
Out[113]:
a    1
b    0
c    3
dtype: int64
```

As usual, the union of the two indices is taken, and non-overlapping values are filled with missing values.

```
In [114]: ser3 = pd.Series([2, 4, 6], index=["b", "c", "d"])

In [115]: ser3
Out[115]:
b    2
c    4
d    6
dtype: int64

In [116]: np.remainder(ser1, ser3)
Out[116]:
a    NaN
b    0.0
c    3.0
d    NaN
dtype: float64
```

When a binary ufunc is applied to a `Series` and `Index`, the Series implementation takes precedence and a Series is returned.

```
In [117]: ser = pd.Series([1, 2, 3])

In [118]: idx = pd.Index([4, 5, 6])

In [119]: np.maximum(ser, idx)
Out[119]:
0    4
1    5
2    6
dtype: int64
```

NumPy ufuncs are safe to apply to `Series` backed by non-ndarray arrays, for example `arrays.SparseArray` (see *Sparse calculation*). If possible, the ufunc is applied without converting the underlying data to an ndarray.

Console display

Very large DataFrames will be truncated to display them in the console. You can also get a summary using `info()`. (Here I am reading a CSV version of the **baseball** dataset from the **plyr** R package):

```
In [120]: baseball = pd.read_csv("data/baseball.csv")

In [121]: print(baseball)
       id     player  year  stint team  lg    g   ab   r    h  X2b  X3b  hr   rbi   sb␣
↪  cs  bb    so  ibb  hbp   sh   sf  gidp
0   88641  womacto01  2006      2  CHN  NL   19   50   6   14    1    0   1   2.0  1.0␣
↪ 1.0   4   4.0  0.0  0.0  3.0  0.0   0.0
1   88643  schilcu01  2006      1  BOS  AL   31    2   0    1    0    0   0   0.0  0.0␣
↪ 0.0   0   1.0  0.0  0.0  0.0  0.0   0.0
..    ...        ...   ...    ...  ...  ..   ..  ...  ..  ...  ...  ...  ..   ...  ...␣
↪ ...  ..   ...  ...  ...  ...  ...   ...
98  89533   aloumo01  2007      1  NYN  NL   87  328  51  112   19    1  13  49.0  3.0␣
↪ 0.0  27  30.0  5.0  2.0  0.0  3.0  13.0
99  89534  alomasa02  2007      1  NYN  NL    8   22   1    3    1    0   0   0.0  0.0␣
↪ 0.0   0   3.0  0.0  0.0  0.0  0.0   0.0

[100 rows x 23 columns]

In [122]: baseball.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 23 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      100 non-null    int64
 1   player  100 non-null    object
 2   year    100 non-null    int64
 3   stint   100 non-null    int64
 4   team    100 non-null    object
 5   lg      100 non-null    object
 6   g       100 non-null    int64
 7   ab      100 non-null    int64
 8   r       100 non-null    int64
 9   h       100 non-null    int64
 10  X2b     100 non-null    int64
 11  X3b     100 non-null    int64
 12  hr      100 non-null    int64
 13  rbi     100 non-null    float64
 14  sb      100 non-null    float64
 15  cs      100 non-null    float64
 16  bb      100 non-null    int64
 17  so      100 non-null    float64
 18  ibb     100 non-null    float64
 19  hbp     100 non-null    float64
 20  sh      100 non-null    float64
 21  sf      100 non-null    float64
 22  gidp    100 non-null    float64
dtypes: float64(9), int64(11), object(3)
memory usage: 18.1+ KB
```

However, using `to_string` will return a string representation of the DataFrame in tabular form, though it won't always fit the console width:

```
In [123]: print(baseball.iloc[-20:, :12].to_string())
       id     player  year  stint team  lg    g   ab   r    h  X2b  X3b
80  89474  finlest01  2007      1  COL  NL   43   94   9   17    3    0
81  89480  embreal01  2007      1  OAK  AL    4    0   0    0    0    0
82  89481  edmonji01  2007      1  SLN  NL  117  365  39   92   15    2
83  89482  easleda01  2007      1  NYN  NL   76  193  24   54    6    0
84  89489  delgaca01  2007      1  NYN  NL  139  538  71  139   30    0
85  89493  cormirh01  2007      1  CIN  NL    6    0   0    0    0    0
86  89494  coninje01  2007      2  NYN  NL   21   41   2    8    2    0
87  89495  coninje01  2007      1  CIN  NL   80  215  23   57   11    1
88  89497  clemero02  2007      1  NYA  AL    2    2   0    1    0    0
89  89498  claytro01  2007      2  BOS  AL    8    6   1    0    0    0
90  89499  claytro01  2007      1  TOR  AL   69  189  23   48   14    0
91  89501  cirilje01  2007      2  ARI  NL   28   40   6    8    4    0
92  89502  cirilje01  2007      1  MIN  AL   50  153  18   40    9    2
93  89521  bondsba01  2007      1  SFN  NL  126  340  75   94   14    0
94  89523  biggicr01  2007      1  HOU  NL  141  517  68  130   31    3
95  89525  benitar01  2007      2  FLO  NL   34    0   0    0    0    0
96  89526  benitar01  2007      1  SFN  NL   19    0   0    0    0    0
97  89530  ausmubr01  2007      1  HOU  NL  117  349  38   82   16    3
98  89533   aloumo01  2007      1  NYN  NL   87  328  51  112   19    1
99  89534  alomasa02  2007      1  NYN  NL    8   22   1    3    1    0
```

Wide DataFrames will be printed across multiple rows by default:

```
In [124]: pd.DataFrame(np.random.randn(3, 12))
Out[124]:
          0         1         2         3         4         5         6         7   ␣
↪   8         9        10        11
0 -1.226825  0.769804 -1.281247 -0.727707 -0.121306 -0.097883  0.695775  0.341734  0.
↪959726 -1.110336 -0.619976  0.149748
1 -0.732339  0.687738  0.176444  0.403310 -0.154951  0.301624 -2.179861 -1.369849 -0.
↪954208  1.462696 -1.743161 -0.826591
2 -0.345352  1.314232  0.690579  0.995761  2.396780  0.014871  3.357427 -0.317441 -1.
↪236269  0.896171 -0.487602 -0.082240
```

You can change how much to print on a single row by setting the `display.width` option:

```
In [125]: pd.set_option("display.width", 40)  # default is 80

In [126]: pd.DataFrame(np.random.randn(3, 12))
Out[126]:
          0         1         2         3         4         5         6         7   ␣
↪   8         9        10        11
0 -2.182937  0.380396  0.084844  0.432390  1.519970 -0.493662  0.600178  0.274230  0.
↪132885 -0.023688  2.410179  1.450520
1  0.206053 -0.251905 -2.213588  1.063327  1.266143  0.299368 -0.863838  0.408204 -1.
↪048089 -0.025747 -0.988387  0.094055
2  1.262731  1.289997  0.082423 -0.055758  0.536580 -0.489682  0.369374 -0.034571 -2.
↪484478 -0.281461  0.030711  0.109121
```

You can adjust the max width of the individual columns by setting `display.max_colwidth`

```
In [127]: datafile = {
   .....:     "filename": ["filename_01", "filename_02"],
   .....:     "path": [
```

(continues on next page)

(continued from previous page)

```
   .....:         "media/user_name/storage/folder_01/filename_01",
   .....:         "media/user_name/storage/folder_02/filename_02",
   .....:     ],
   .....: }
   .....:

In [128]: pd.set_option("display.max_colwidth", 30)

In [129]: pd.DataFrame(datafile)
Out[129]:
      filename                           path
0  filename_01  media/user_name/storage/fo...
1  filename_02  media/user_name/storage/fo...

In [130]: pd.set_option("display.max_colwidth", 100)

In [131]: pd.DataFrame(datafile)
Out[131]:
      filename                                         path
0  filename_01  media/user_name/storage/folder_01/filename_01
1  filename_02  media/user_name/storage/folder_02/filename_02
```

You can also disable this feature via the `expand_frame_repr` option. This will print the table in one block.

DataFrame column attribute access and IPython completion

If a DataFrame column label is a valid Python variable name, the column can be accessed like an attribute:

```
In [132]: df = pd.DataFrame({"foo1": np.random.randn(5), "foo2": np.random.randn(5)})

In [133]: df
Out[133]:
       foo1      foo2
0  1.126203  0.781836
1 -0.977349 -1.071357
2  1.474071  0.441153
3 -0.064034  2.353925
4 -1.282782  0.583787

In [134]: df.foo1
Out[134]:
0    1.126203
1   -0.977349
2    1.474071
3   -0.064034
4   -1.282782
Name: foo1, dtype: float64
```

The columns are also connected to the IPython completion mechanism so they can be tab-completed:

```
In [5]: df.foo<TAB>  # noqa: E225, E999
df.foo1  df.foo2
```