NumPy Tutorial: Data Analysis with Python

NumPy is a commonly used Python data analysis package. By using NumPy, you can speed up your workflow, and interface with other packages in the Python ecosystem, like scikit-learn, that use NumPy under the hood. NumPy was originally developed in the mid 2000s, and arose from an even older package called Numeric. This longevity means that almost every data analysis or machine learning package for Python leverages NumPy in some way. **NumPy** library is an important foundational tool for studying **Machine Learning**. Many of its functions are very **useful** for performing any mathematical or scientific calculation. As it is known that mathematics is the foundation of **machine learning**, most of the mathematical tasks can be performed using **NumPy**.
In this tutorial, we'll walk through using NumPy to analyze data on wine quality. The data contains information on various attributes of wines, such as pH and fixed acidity, along with a quality score between 0 and 10 for each wine. The quality score is the average of at least 3 human taste testers. As we learn how to work with NumPy, we'll try to figure out more about the perceived quality of wine.

The wines we'll be analyzing are from the Minho region of Portugal.
The data was downloaded from the UCI Machine Learning Repository, and is available here. Here are the first few rows of the winequality-red.csv file, which we'll be using throughout this tutorial:

```
"fixed acidity";"volatile acidity";"citric acid";"residual sugar";"chlorides";"free sulfur dioxide";"total sulfur dioxide";"density";"pH";"sulphates";"alcohol";"quality"

7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5


7.8;0.88;0;2.6;0.098;25;67;0.9968;3.2;0.68;9.8;5
```

The data is in what I'm going to call *ssv* (semicolon separated values) format — each record is separated by a semicolon (;), and rows are separated by a new line. There are 1600 rows in the file, including a header row, and 12 columns.
Before we get started, a quick version note — we'll be using Python 3.5. Our code examples will be done using Jupyter notebook.

Lists Of Lists for CSV Data

Before using NumPy, we'll first try to work with the data using Python and the csv package. We can read in the file using the csv.reader object, which will allow us to read in and split up all the content from the *ssv* file.
In the below code, we:

- Import the csv library.
- Open the winequality-red.csv file.
    - With the file open, create a new csv.reader object.

        ▪ Pass in the keyword argument `delimiter=";"` to make sure that the records are split up on the semicolon character instead of the default comma character.
    ◦ Call the list type to get all the rows from the file.
    ◦ Assign the result to `wines`.

```
import csv

with open('winequality-red.csv', 'r') as f:


    wines = list(csv.reader(f, delimiter=';'))
```

Once we’ve read in the data, we can print out the first `3` rows:

```
print(wines[:3])
[['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density', 'pH',
'sulphates', 'alcohol', 'quality'], ['7.4', '0.7', '0', '1.9', '0.076', '11',
'34', '0.9978', '3.51', '0.56', '9.4', '5'], ['7.8', '0.88', '0', '2.6', '0.098',
'25', '67', '0.9968', '3.2', '0.68', '9.8', '5']]
```

The data has been read into a list of lists. Each inner list is a row from the *ssv* file. As you may have noticed, each item in the entire list of lists is represented as a string, which will make it harder to do computations.
We’ll format the data into a table to make it easier to view:

fixed acidity	**volatile acidity**	**citric acid**	**residual sugar**	**chlorides**	**free sulfur dioxide**	**total sulfur dioxide**	**density**	**pH**	**sulphates**	**alcohol**
7.4	0.70	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4
7.8	0.88	0	2.6	0.098	25	67	0.9968	3.20	0.68	9.8

As you can see from the table above, we’ve read in three rows, the first of which contains column headers. Each row after the header row represents a wine. The first element of each row is the `fixed acidity`, the second is the `volatile acidity`, and so on. We can find the average `quality` of the wines. The below code will:

- Extract the last element from each row after the header row.

- Convert each extracted element to a float.
- Assign all the extracted elements to the list `qualities`.
- Divide the sum of all the elements in `qualities` by the total number of elements in `qualities` to the get the mean.

```
qualities =

[float(item[-1]) for item in wines[1:]]


sum(qualities) / len(qualities)

5.6360225140712945
```

Although we were able to do the calculation we wanted, the code is fairly complex, and it won't be fun to have to do something similar every time we want to compute a quantity. Luckily, we can use NumPy to make it easier to work with our data.

Numpy 2-Dimensional Arrays

With NumPy, we work with multidimensional arrays. We'll dive into all of the possible types of multidimensional arrays later on, but for now, we'll focus on 2-dimensional arrays. A 2-dimensional array is also known as a matrix, and is something you should be familiar with. In fact, it's just a different way of thinking about a list of lists. A matrix has rows and columns. By specifying a row number and a column number, we're able to extract an element from a matrix.

In the below matrix, the first row is the header row, and the first column is the `fixed acidity` column:

fixed acidity	**volatile acidity**	**citric acid**	**residual sugar**	**chlorides**	**free sulfur dioxide**	**total sulfur dioxide**	**density**	**pH**	**sulphates**	**alcohol**
7.4	0.70	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4
7.8	0.88	0	2.6	0.098	25	67	0.9968	3.20	0.68	9.8

If we picked the element at the first row and the second column, we'd get `volatile acidity`. If we picked the element in the third row and the second column, we'd get `0.88`.

In a NumPy array, the number of dimensions is called the rank, and each dimension is called an axis. So the rows are the first axis, and the columns are the second axis.

Now that you understand the basics of matrices, let's see how we can get from our list of lists to a NumPy array.

Creating A NumPy Array

We can create a NumPy array using the numpy.array function. If we pass in a list of lists, it will automatically create a NumPy array with the same number of rows and columns. Because we want all of the elements in the array to be *float* elements for easy computation, we'll leave off the header row, which contains *strings*. One of the limitations of NumPy is that all the elements in an array have to be of the same type, so if we include the header row, all the elements in the array will be read in as strings. Because we want to be able to do computations like find the average `quality` of the wines, we need the elements to all be floats.

In the below code, we:

- Import the `numpy` package.
- Pass the list of lists `wines` into the `array` function, which converts it into a NumPy array.
    - Exclude the header row with list slicing.
    - Specify the keyword argument `dtype` to make sure each element is converted to a float. We'll dive more into what the `dtype` is later on.

```
import csv

with open("winequality-red.csv", 'r') as f:

    wines = list(csv.reader(f, delimiter=";"))

import numpy as np


wines = np.array(wines[1:], dtype=np.float)
```

Try running the above code and seeing what happens!

If we display `wines`, we'll now get a NumPy array:

```
wines
array([[ 7.4 , 0.7 , 0. , ..., 0.56 , 9.4 , 5. ],

[ 7.8 , 0.88 , 0. , ..., 0.68 , 9.8 , 5. ],

[ 7.8 , 0.76 , 0.04 , ..., 0.65 , 9.8 , 5. ],

...,

[ 6.3 , 0.51 , 0.13 , ..., 0.75 , 11. , 6. ],

[ 5.9 , 0.645, 0.12 , ..., 0.71 , 10.2 , 5. ],
[ 6.  , 0.31 , 0.47 , ..., 0.66 , 11.  , 6.  ]])
```

We can check the number of rows and columns in our data using the shape property of NumPy arrays:

```
wines.shape

(1599, 12)
```

Alternative NumPy Array Creation Methods

There are a variety of methods that you can use to create NumPy arrays. To start with, you can create an array where every element is zero. The below code will create an array with `3` rows and `4` columns, where every element is `0`, using numpy.zeros:

```
import numpy as np

empty_array = np.zeros((3,4))




empty_array
```

It's useful to create an array with all zero elements in cases when you need an array of fixed size, but don't have any values for it yet.

You can also create an array where each element is a random number using numpy.random.rand. Here's an example:

```
np.random.rand(3,4)
array([[ 0.2247223 , 0.92240549, 0.14541893, 0.61731257],

[ 0.00154957, 0.82342197, 0.74044906, 0.11466845],


[ 0.6152478 , 0.14433138, 0.13009583, 0.22981301]])
```

Creating arrays full of random numbers can be useful when you want to quickly test your code with sample arrays.

Using NumPy To Read In Files

It's possible to use NumPy to directly read csv or other files into arrays. We can do this using the numpy.genfromtxt function. We can use it to read in our initial data on red wines.

In the below code, we:

- Use the `genfromtxt` function to read in the `winequality-red.csv` file.
- Specify the keyword argument `delimiter=";"` so that the fields are parsed properly.

- Specify the keyword argument `skip_header=1` so that the header row is skipped.

```
wines = np.genfromtxt("winequality-red.csv", delimiter=";", skip_header=1)
```

`wines` will end up looking the same as if we read it into a list then converted it to an array of floats. NumPy will automatically pick a data type for the elements in an array based on their format.

Indexing NumPy Arrays

We now know how to create arrays, but unless we can retrieve results from them, there isn't a lot we can do with NumPy. We can use array indexing to select individual elements, groups of elements, or entire rows and columns. One important thing to keep in mind is that just like Python lists, NumPy is zero-indexed, meaning that the index of the first row is `0`, and the index of the first column is `0`. If we want to work with the fourth row, we'd use index `3`, if we want to work with the second row, we'd use index `1`, and so on. We'll again work with the `wines` array:

7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4
7.8	0.88	0.00	2.6	0.098	25	67	0.9968	3.20	0.68	9.8
7.8	0.76	0.04	2.3	0.092	15	54	0.9970	3.26	0.65	9.8
11.2	0.28	0.56	1.9	0.075	17	60	0.9980	3.16	0.58	9.8
7.4	0.70	0.00	1.9	0.076	11	34	0.9978	3.51	0.56	9.4

Let's select the element at row `3` and column `4`. In the below code, we pass in the index `2` as the row index, and the index `3` as the column index. This retrieves the value from the fourth column of the third row:

```
wines[2,3]

2.2999999999999998
```

Since we're working with a 2-dimensional array in NumPy, we specify 2 indexes to retrieve an element. The first index is the row, or axis `1`, index, and the second index is the column, or axis `2`, index. Any element in `wines` can be retrieved using `2` indexes.

Slicing NumPy Arrays

If we instead want to select the first three items from the fourth column, we can do it using a colon (`:`). A colon indicates that we want to select all the elements from the starting index up to but not including the ending index. This is also known as a slice:

```
wines[0:3,3]

array([ 1.9, 2.6, 2.3])
```

Just like with list slicing, it's possible to omit the `0` to just retrieve all the elements from the beginning up to element `3`:

```
wines[:3,3]

array([ 1.9, 2.6, 2.3])
```

We can select an entire column by specifying that we want all the elements, from the first to the last. We specify this by just using the colon (`:`), with no starting or ending indices. The below code will select the entire fourth column:

```
wines[:,3]

array([ 1.9, 2.6, 2.3, ..., 2.3, 2. , 3.6])
```

We selected an entire column above, but we can also extract an entire row:

```
wines[3,:]
array([ 11.2 , 0.28 , 0.56 , 1.9 , 0.075, 17. , 60.,


0.998, 3.16 , 0.58 , 9.8 , 6. ])
```

If we take our indexing to the extreme, we can select the entire array using two colons to select all the rows and columns in `wines`. This is a great party trick, but doesn't have a lot of good applications:

```
wines[:,:]
array([[ 7.4 , 0.7 , 0. , ..., 0.56 , 9.4 , 5. ],

[ 7.8 , 0.88 , 0. , ..., 0.68 , 9.8 , 5. ],<br />

[ 7.8 , 0.76 , 0.04 , ..., 0.65 , 9.8 , 5. ],<br />

...,

[ 6.3 , 0.51 , 0.13 , ..., 0.75 , 11. , 6. ],

[ 5.9 , 0.645, 0.12 , ..., 0.71 , 10.2 , 5. ],<br />


[ 6. , 0.31 , 0.47 , ..., 0.66 , 11. , 6. ]])
```

Assigning Values To NumPy Arrays

We can also use indexing to assign values to certain elements in arrays. We can do this by assigning directly to the indexed value:

```
wines[1,5] = 10
```

We can do the same for slices. To overwrite an entire column, we can do this:

```
wines[:,10] = 50
```

The above code overwrites all the values in the eleventh column with `50`.

1-Dimensional NumPy Arrays

So far, we've worked with 2-dimensional arrays, such as `wines`. However, NumPy is a package for working with multidimensional arrays. One of the most common types of multidimensional arrays is the 1-dimensional array, or vector. As you may have noticed above, when we sliced `wines`, we retrieved a 1-dimensional array. A 1-dimensional array only needs a single index to retrieve an element. Each row and column in a 2-dimensional array is a 1-dimensional array. Just like a list of lists is analogous to a 2-dimensional array, a single list is analogous to a 1-dimensional array. If we slice wines and only retrieve the third row, we get a 1-dimensional array:

```
third_wine = wines[3,:]
```

Here's how `third_wine` looks:

```
11.200

0.280

0.560

1.900

0.075

17.000

60.000

0.998

3.160

0.580

9.800


6.000
```

We can retrieve individual elements from `third_wine` using a single index. The below code will display the second item in `third_wine`:

```
third_wine[1]

0.28000000000000003
```

Most NumPy functions that we've worked with, such as numpy.random.rand, can be used with multidimensional arrays. Here's how we'd use `numpy.random.rand` to generate a random vector:

```
np.random.rand(3)

array([ 0.88588862, 0.85693478, 0.19496774])
```

Previously, when we called `np.random.rand`, we passed in a shape for a 2-dimensional array, so the result was a 2-dimensional array. This time, we passed in a shape for a single dimensional array. The shape specifies the number of dimensions, and the size of the array in each dimension. A shape of `(10,10)` will be a 2-dimensional array with `10` rows and `10` columns. A shape of `(10,)` will be a 1-dimensional array with `10` elements.

Where NumPy gets more complex is when we start to deal with arrays that have more than `2` dimensions.

N-Dimensional NumPy Arrays

This doesn't happen extremely often, but there are cases when you'll want to deal with arrays that have greater than `3` dimensions. One way to think of this is as a list of lists of lists. Let's say we want to store the monthly earnings of a store, but we want to be able to quickly lookup the results for a quarter, and for a year. The earnings for one year might look like this:

```
[500, 505, 490, 810, 450, 678, 234, 897, 430, 560, 1023, 640]
```

The store earned `$500` in January, `$505` in February, and so on. We can split up these earnings by quarter into a list of lists:

```
year_one = [

    [500,505,490],

    [810,450,678],

    [234,897,430],

    [560,1023,640]


]
```

We can retrieve the earnings from January by calling `year_one[0][0]`. If we want the results for a whole quarter, we can call `year_one[0]` or `year_one[1]`. We now have a 2-dimensional array, or matrix. But what if we now want to add the results from another year? We have to add a third dimension:

```
earnings = [
    [

        [500,505,490],

        [810,450,678],

        [234,897,430],

        [560,1023,640]

    ],

    [

        [600,605,490],

        [345,900,1000],

        [780,730,710],

        [670,540,324]

    ]


]
```

We can retrieve the earnings from January of the first year by calling `earnings[0][0][0]`. We now need three indexes to retrieve a single element. A three-dimensional array in NumPy is much the same. In fact, we can convert `earnings` to an array and then get the earnings for January of the first year:

```
earnings = np.array(earnings)


earnings[0,0,0]

500
```

We can also find the shape of the array:

```
earnings.shape

(2, 4, 3)
```

Indexing and slicing work the exact same way with a 3-dimensional array, but now we have an extra axis to pass in. If we wanted to get the earnings for January of all years, we could do this:

```
earnings[:,0,0]

array([500, 600])
```

If we wanted to get first quarter earnings from both years, we could do this:

```
earnings[:,0,:]
```

```
array([[500, 505, 490],


[600, 605, 490]])
```

Adding more dimensions can make it much easier to query your data if it's organized in a certain way. As we go from 3-dimensional arrays to 4-dimensional and larger arrays, the same properties apply, and they can be indexed and sliced in the same ways.

NumPy Data Types

As we mentioned earlier, each NumPy array can store elements of a single data type. For example, `wines` contains only float values. NumPy stores values using its own data types, which are distinct from Python types like `float` and `str`. This is because the core of NumPy is written in a programming language called C, which stores data differently than the Python data types. NumPy data types map between Python and C, allowing us to use NumPy arrays without any conversion hitches.
You can find the data type of a NumPy array by accessing the dtype property:

```
wines.dtype

dtype('float64')
```

NumPy has several different data types, which mostly map to Python data types, like `float`, and `str`. You can find a full listing of NumPy data types here, but here are a few important ones:

- `float` — numeric floating point data.
- `int` — integer data.
- `string` — character data.
- `object` — Python objects.

Data types additionally end with a suffix that indicates how many bits of memory they take up. So `int32` is a 32 bit integer data type, and `float64` is a `64` bit float data type.

Converting Data Types

You can use the numpy.ndarray.astype method to convert an array to a different type. The method will actually copy the array, and return a new array with the specified data type. For instance, we can convert `wines` to the `int` data type:

```
wines.astype(int)
array([[ 7, 0, 0, ..., 0, 9, 5],

[ 7, 0, 0, ..., 0, 9, 5],

[ 7, 0, 0, ..., 0, 9, 5],

...,

[ 6, 0, 0, ..., 0, 11, 6],
[ 5, 0, 0, ..., 0, 10, 5],


[ 6, 0, 0, ..., 0, 11, 6]])
```

As you can see above, all of the items in the resulting array are integers. Note that we used the Python `int` type instead of a NumPy data type when converting `wines`. This is because several Python data types, including `float`, `int`, and `string`, can be used with NumPy, and are automatically converted to NumPy data types.

We can check the [name](https://docs.scipy.org/doc/numpy/reference/generated/numpy.dtype.name.html) property of the [dtype](https://docs.scipy.org/doc/numpy/reference/generated/numpy.dtype.html) of the resulting array to see what data type NumPy mapped the resulting array to:

```
int_wines = wines.astype(int)


int_wines.dtype.name

'int64'
```

The array has been converted to a 64-bit integer data type. This allows for very long integer values, but takes up more space in memory than storing the values as 32-bit integers.

If you want more control over how the array is stored in memory, you can directly create NumPy dtype objects like [numpy.int32](https://docs.scipy.org/doc/numpy/user/basics.types.html):

```
np.int32

numpy.int32
```

You can use these directly to convert between types:

```
wines.astype(np.int32)
array([[ 7, 0, 0, ..., 0, 9, 5],

[ 7, 0, 0, ..., 0, 9, 5],

[ 7, 0, 0, ..., 0, 9, 5],

...,

[ 6, 0, 0, ..., 0, 11, 6],

[ 5, 0, 0, ..., 0, 10, 5],


[ 6, 0, 0, ..., 0, 11, 6]], dtype=int32)
```

NumPy Array Operations

NumPy makes it simple to perform mathematical operations on arrays. This is one of the primary advantages of NumPy, and makes it quite easy to do computations.

Single Array Math

If you do any of the basic mathematical operations (`/`, `*`, `-`, `+`, `^`) with an array and a value, it will apply the operation to each of the elements in the array.

Let’s say we want to add `10` points to each quality score because we’re drunk and feeling generous. Here’s how we’d do that:

```
wines[:,11] + 10

array([ 15., 15., 15., ..., 16., 15., 16.])
```

Note that the above operation won’t change the `wines` array — it will return a new 1-dimensional array where `10` has been added to each element in the quality column of wines.

If we instead did `+=`, we’d modify the array in place:

```
wines[:,11] += 10


wines[:,11]

array([ 15., 15., 15., ..., 16., 15., 16.])
```

All the other operations work the same way. For example, if we want to multiply each of the quality score by `2`, we could do it like this:

```
wines[:,11] * 2

array([ 30., 30., 30., ..., 32., 30., 32.])
```

Multiple Array Math

It’s also possible to do mathematical operations between arrays. This will apply the operation to pairs of elements. For example, if we add the `quality` column to itself, here’s what we get:

```
wines[:,11] + wines[:,11]

array([ 10., 10., 10., ..., 12., 10., 12.])
```

Note that this is equivalent to `wines[11] * 2` — this is because NumPy adds each pair of elements. The first element in the first array is added to the first element in the second array, the second to the second, and so on.

We can also use this to multiply arrays. Let’s say we want to pick a wine that maximizes alcohol content and quality (we want to get drunk, but we’re classy). We’d multiply `alcohol` by `quality`, and select the wine with the highest score:

```
wines[:,10] * wines[:,11]

array([ 47., 49., 49., ..., 66., 51., 66.])
```

All of the common operations (`/`, `*`, `-`, `+`, `^`) will work between arrays.

Broadcasting

Unless the arrays that you’re operating on are the exact same size, it’s not possible to do elementwise operations. In cases like this, NumPy performs broadcasting to try to match up elements. Essentially, broadcasting involves a few steps:

- The last dimension of each array is compared.
  - If the dimension lengths are equal, or one of the dimensions is of length `1`, then we keep going.

    - If the dimension lengths aren't equal, and none of the dimensions have length `1`, then there's an error.
- Continue checking dimensions until the shortest array is out of dimensions.

For example, the following two shapes are compatible:

```
A: (50,3)


B (3,)
```

This is because the length of the trailing dimension of array `A` is `3`, and the length of the trailing dimension of array `B` is `3`. They're equal, so that dimension is okay. Array `B` is then out of elements, so we're okay, and the arrays are compatible for mathematical operations.

The following two shapes are also compatible:

```
A: (1,2)


B (50,2)
```

The last dimension matches, and `A` is of length `1` in the first dimension.

These two arrays don't match:

```
A: (50,50)


B: (49,49)
```

The lengths of the dimensions aren't equal, and neither array has either dimension length equal to `1`.

There's a detailed explanation of broadcasting here, but we'll go through a few examples to illustrate the principle:

```
wines * np.array([1,2])
-----------------------------------------------------------------------


ValueError Traceback (most recent call last)

<ipython -input-40-821086ccaf65> in <module>()

----> 1 wines * np.array([1,2])


ValueError: operands could not be broadcast together with shapes (1599,12)

(2,)</module></ipython>
```

The above example didn't work because the two arrays don't have a matching trailing dimension. Here's an example where the last dimension does match:

```
array_one = np.array(


    [


        [1,2],

        [3,4]

    ]

)

array_two = np.array([4,5])


array_one + array_two
array([[5, 7],


[7, 9]])
```

As you can see, `array_two` has been broadcasted across each row of `array_one`. Here's an example with our `wines` data:

```
rand_array = np.random.rand(12)


wines + rand_array
array([[ 8.08375389, 0.89047394, 0.77022918, ..., 0.94917479,

10.34668852, 5.34569289],

[ 8.48375389, 1.07047394, 0.77022918, ..., 1.06917479,

10.74668852, 5.34569289],

[ 8.48375389, 0.95047394, 0.81022918, ..., 1.03917479,

10.74668852, 5.34569289],

...,

[ 6.98375389, 0.70047394, 0.90022918, ..., 1.13917479,

11.94668852, 6.34569289],

[ 6.58375389, 0.83547394, 0.89022918, ..., 1.09917479,

11.14668852, 5.34569289],

[ 6.68375389, 0.50047394, 1.24022918, ..., 1.04917479,


11.94668852, 6.34569289]])
```

Elements of `rand_array` are broadcast over each row of `wines`, so the first column of `wines` has the first value in `rand_array` added to it, and so on.

NumPy Array Methods

In addition to the common mathematical operations, NumPy also has several methods that you can use for more complex calculations on arrays. An example of this is the numpy.ndarray.sum method. This finds the sum of all the elements in an array by default:

```
wines[:,11].sum()

9012.0
```

The total of all of our quality ratings is `154.1788`. We can pass the `axis` keyword argument into the `sum` method to find sums over an axis. If we call `sum` across the `wines` matrix, and pass in `axis=0`, we'll find the sums over the first axis of the array. This will give us the sum of all the values in every column. This may seem backwards that the sums over the first axis would give us the sum of each column, but one way to think about this is that the specified axis is the one "going away". So if we specify `axis=0`, we want the rows to go away, and we want to find the sums for each of the remaining axes across each row:

```
wines.sum(axis=0)
array([ 13303.1 , 843.985 , 433.29 , 4059.55 ,

139.859 , 25384. , 74302. , 1593.79794,


5294.47 , 1052.38 , 16666.35 , 9012. ])
```

We can verify that we did the sum correctly by checking the shape. The shape should be `12`, corresponding to the number of columns:

```
wines.sum(axis=0).shape

(12,)
```

If we pass in `axis=1`, we'll find the sums over the second axis of the array. This will give us the sum of each row:

```
wines.sum(axis=1)
array([ 74.5438 , 123.0548 , 99.699 , ...,<br />

100.48174, 105.21547,


92.49249])
```

There are several other methods that behave like the `sum` method, including:

- numpy.ndarray.mean — finds the mean of an array.
- numpy.ndarray.std — finds the standard deviation of an array.
- numpy.ndarray.min — finds the minimum value in an array.
- numpy.ndarray.max — finds the maximum value in an array.

You can find a full list of array methods here.

NumPy Array Comparisons

NumPy makes it possible to test to see if rows match certain values using mathematical comparison operations like `<`, `>`, `>=`, `<=`, and `==`. For example, if we want to see which wines have a quality rating higher than `5`, we can do this:

```
wines[:,11] > 5

array([False, False, False, ..., True, False, True], dtype=bool)
```

We get a Boolean array that tells us which of the wines have a quality rating greater than `5`. We can do something similar with the other operators. For instance, we can see if any wines have a quality rating equal to `10`:

```
wines[:,11] == 10

array([False, False, False, ..., False, False, False], dtype=bool)
```

Subsetting

One of the powerful things we can do with a Boolean array and a NumPy array is select only certain rows or columns in the NumPy array. For example, the below code will only select rows in `wines` where the quality is over `7`:

```
high_quality = wines[:,11] > 7


wines[high_quality,:][:3,:]
array([[ 7.90000000e+00, 3.50000000e-01, 4.60000000e-01,

3.60000000e+00, 7.80000000e-02, 1.50000000e+01,

3.70000000e+01, 9.97300000e-01, 3.35000000e+00,

8.60000000e-01, 1.28000000e+01, 8.00000000e+00],

[ 1.03000000e+01, 3.20000000e-01, 4.50000000e-01,

6.40000000e+00, 7.30000000e-02, 5.00000000e+00,

1.30000000e+01, 9.97600000e-01, 3.23000000e+00,

8.20000000e-01, 1.26000000e+01, 8.00000000e+00],

[ 5.60000000e+00, 8.50000000e-01, 5.00000000e-02,

1.40000000e+00, 4.50000000e-02, 1.20000000e+01,

8.80000000e+01, 9.92400000e-01, 3.56000000e+00,


8.20000000e-01, 1.29000000e+01, 8.00000000e+00]])
```

We select only the rows where `high_quality` contains a `True` value, and all of the columns. This subsetting makes it simple to filter arrays for certain criteria. For example, we can look for wines with a lot of alcohol and high quality. In order to specify multiple conditions, we have to place each condition in parentheses, and separate conditions with an ampersand (`&`):

```
high_quality_and_alcohol = (wines[:,10] > 10) & (wines[:,11] > 7)


wines[high_quality_and_alcohol,10:]
array([[ 12.8, 8. ],

[ 12.6, 8. ],

[ 12.9, 8. ],

[ 13.4, 8. ],

[ 11.7, 8. ],

[ 11. , 8. ],

[ 11. , 8. ],

[ 14. , 8. ],

[ 12.7, 8. ],

[ 12.5, 8. ],

[ 11.8, 8. ],

[ 13.1, 8. ],

[ 11.7, 8. ],

[ 14. , 8. ],

[ 11.3, 8. ],


[ 11.4, 8. ]])
```

We can combine subsetting and assignment to overwrite certain values in an array:

```
high_quality_and_alcohol = (wines[:,10] > 10) & (wines[:,11] > 7)


wines[high_quality_and_alcohol,10:] = 20
```

Reshaping NumPy Arrays

We can change the shape of arrays while still preserving all of their elements. This often can make it easier to access array elements. The simplest reshaping is to flip the axes, so rows become columns, and vice versa. We can accomplish this with the numpy.transpose function:

```
np.transpose(wines).shape

(12, 1599)
```

We can use the numpy.ravel function to turn an array into a one-dimensional representation. It will essentially flatten an array into a long sequence of values:

```
wines.ravel()

array([ 7.4 , 0.7 , 0. , ..., 0.66, 11. , 6. ])
```

Here's an example where we can see the ordering of `numpy.ravel`:

```
array_one = np.array(

    [

        [1, 2, 3, 4],

        [5, 6, 7, 8]

    ]

)


array_one.ravel()

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

Finally, we can use the numpy.reshape function to reshape an array to a certain shape we specify. The below code will turn the second row of `wines` into a 2-dimensional array with `2` rows and `6` columns:

```
wines[1,:].reshape((2,6))
array([[ 7.8 , 0.88 , 0. , 2.6 , 0.098 , 25. ],


[ 67. , 0.9968, 3.2 , 0.68 , 9.8 , 5. ]])
```

Combining NumPy Arrays

With NumPy, it's very common to combine multiple arrays into a single unified array. We can use numpy.vstack to vertically stack multiple arrays. Think of it like the second arrays's items being added as new rows to the first array. We can read in the `winequality-white.csv` dataset that contains information on the quality of white wines, then combine it with our existing dataset, `wines`, which contains information on red wines.

In the below code, we:

- Read in `winequality-white.csv`.
- Display the shape of `white_wines`.

```
white_wines     =     np.genfromtxt("winequality-white.csv",     delimiter=";",
skip_header=1)


white_wines.shape

(4898, 12)
```

As you can see, we have attributes for `4898` wines. Now that we have the white wines data, we can combine all the wine data.
In the below code, we:

- Use the `vstack` function to combine `wines` and `white_wines`.
- Display the shape of the result.

```
all_wines = np.vstack((wines, white_wines))


all_wines.shape

(6497, 12)
```

As you can see, the result has `6497` rows, which is the sum of the number of rows in `wines` and the number of rows in `red_wines`.
If we want to combine arrays horizontally, where the number of rows stay constant, but the columns are joined, then we can use the numpy.hstack function. The arrays we combine need to have the same number of rows for this to work.
Finally, we can use numpy.concatenate as a general purpose version of `hstack` and `vstack`. If we want to concatenate two arrays, we pass them into `concatenate`, then specify the `axis` keyword argument that we want to concatenate along. Concatenating along the first axis is similar to `vstack`, and concatenating along the second axis is similar to `hstack`:

```
np.concatenate((wines, white_wines), axis=0)
array([[ 7.4 , 0.7 , 0. , ..., 0.56, 9.4 , 5. ],

[ 7.8 , 0.88, 0. , ..., 0.68, 9.8 , 5. ],

[ 7.8 , 0.76, 0.04, ..., 0.65, 9.8 , 5. ],

...,

[ 6.5 , 0.24, 0.19, ..., 0.46, 9.4 , 6. ],

[ 5.5 , 0.29, 0.3 , ..., 0.38, 12.8 , 7. ],


[ 6. , 0.21, 0.38, ..., 0.32, 11.8 , 6. ]])
```

Free NumPy Cheat Sheet

If you're interested in learning more about NumPy, check out our interactive course on NumPy and Pandas. You can register and do the first missions for free.
You also might like to take your NumPy skills to the next level with our free NumPy cheat sheet!

Further Reading

You should now have a good grasp of NumPy, and how to apply it to a data set.
If you want to dive into more depth, here are some resources that may be helpful:

- NumPy Quickstart — has good code examples and covers most basic NumPy functionality.
- Python NumPy Tutorial — a great tutorial on NumPy and other Python libraries.
- Visual NumPy Introduction — a guide that uses the game of life to illustrate NumPy concepts.

In our next tutorial, we dive more into Pandas, a library that builds on NumPy and makes data analysis even easier. It solves two of the biggest pain points which are that:

- You can't mix multiple data types in an array.
- You have to remember what type of data each column contains.