Module 3 - Statistics Essentials and Data Pre-processing with Python

This module outlines the techniques for effectively loading, processing and analyzing data in Python. In this process, we will also look at some of the preliminary statistics necessary to understand about the measures of centrality and variance of data. Datasets come from a wide range of sources and various formats, collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of numbers.

For example, text can be converted in various ways into numerical representations, perhaps binary digits representing the frequency of certain words or pairs of words. No matter what the data are, the first step in making it analyzable will be to transform them into arrays of numbers. For this reason, efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science. In this module, we will go through the specialized tools that Python has for handling such numerical arrays: the NumPy package, and the Pandas package.

Introduction to NumPy

This notebook will cover NumPy in detail. NumPy (short for Numerical Python) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python's built-in list type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

Topics Covered:

- Understanding Array data type
- Basics of Numpy Arrays
- Computations on Numpy Arrays: Universal Functions
- Aggregations: Min, Max, and Everything In Between

If you followed the installation instructions for the course packages in your environment, you should have NumPy installed. Go ahead and try to import it. By convention, you'll find that most people in the scientific Python and Data Science communities will import NumPy using `np` as an alias. Throughout this course, you'll find that this is the way we will import and use NumPy.

In [53]:
```
import numpy as np
np.__version__     # command to check the version of package installed
```
Out[53]:
```
'1.18.5'
```

Reminder about Built In Documentation

As you work through this notebook, remember that Jupyter/IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature), as well as the documentation of various functions (using the `?` character).

For example, to display all the contents of the numpy namespace, you can type this:

```
In [3]: np.<TAB>
```

And to display NumPy's built-in documentation, you can use this:

```
In [4]: np?
```

Understanding Array Data Type

Remember the list data type in Python? It can be used to store a collection of data values, each of them belonging to a single data type or multiple data types. Storing and accessing a list comprised of multiple data types comes at a cost. Further, when working with datasets it is more often that we see a collection of data values of a single data type.

Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in `array` module is one such data type that can be used to create dense arrays of a uniform type.

In [54]:
```
import array
L = list(range(10))
A = array.array('i', L)
A
```
Out[54]:
```
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Here `'i'` is a type code indicating the contents are integers.

Much more useful, however, is the `ndarray` object of the NumPy package. While Python's `array` object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data. We will explore these operations in later sections; here we'll demonstrate several ways of creating a NumPy array.

Creating NumPy Arrays from Python Lists

First, we can use `np.array` to create arrays from Python lists:

In [55]:
```
# integer array:
np.array([1, 4, 2, 5, 3])
```
Out[55]:
```
array([1, 4, 2, 5, 3])
```

Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point):

In [56]:
```
np.array([3.14, 4, 2, 3])
```
Out[56]:
```
array([3.14, 4.  , 2.  , 3.  ])
```

If we want to explicitly set the data type of the resulting array, we can use the `dtype` keyword:

In [57]:
```
np.array([1, 2, 3, 4], dtype='float32')
```
Out[57]:
```
array([1., 2., 3., 4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multi-dimensional; here's one way of initializing a multidimensional array using a list of lists:

In [58]:
```
# nested lists result in multi-dimensional arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])
```
Out[58]:
```
array([[2, 3, 4],
       [4, 5, 6],
       [6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

In [59]:
```
# Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)
```
Out[59]:
```
array([[3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14]])
```

In [60]:
```
# Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range() function)
np.arange(0, 20, 2)
```
Out[60]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [61]:
```
# Create a 3x3 array of uniformly distributed
# random values between 0 and 1
np.random.random((3, 3))
```
Out[61]:
```
array([[0.71125897, 0.16526245, 0.43008959],
       [0.61228252, 0.37038403, 0.06785738],
       [0.48412739, 0.7476983 , 0.47720923]])
```

In [62]:
```
# Create a 3x3 array of random integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
```
Out[62]:
```
array([[5, 9, 3],
       [9, 5, 6],
       [5, 1, 0]])
```

The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas are built around the NumPy array. This section will present several examples of using NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays. Get to know them well!

We'll cover a few categories of basic array manipulations here:

- *Attributes of arrays*: Determining the size, shape, memory consumption, and data types of arrays
- *Indexing of arrays*: Getting and setting the value of individual array elements
- *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
- *Reshaping of arrays*: Changing the shape of a given array
- *Joining and splitting of arrays*: Combining multiple arrays into one, and splitting one array into many

NumPy Array Attributes

First let's discuss some useful array attributes. We'll start by defining three random arrays, a one-dimensional, two-dimensional, and three-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated each time this code is run:

In [63]:
```
import numpy as np
np.random.seed(0)  # seed for reproducibility

x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array). Another useful attribute is the `dtype`, the data type of the array.

In [64]:
```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
print("dtype:", x3.dtype)
```
```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
dtype: int32
```

Array Indexing: Accessing Single Elements

If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, the i^{th} value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists:

In [65]:
```
x1
```
Out[65]:
```
array([5, 0, 3, 3, 7, 9])
```

In [66]:
```
x1[1]  # indexing starts with 0 in Python, so index 1 is the second element in an array
```
Out[66]:
```
0
```

In [67]:
```
x1[-1]  # use negative indices to index from the end of the array
```
Out[67]:
```
9
```

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices:

In [68]:
```
x2
```
Out[68]:
```
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

In [69]:
```
x2[2, 0]  # first value indicates row index and second value indicates column index
```
Out[69]:
```
1
```

In [70]:
```
x2[2, -1]
```
Out[70]:
```
7
```

Values can also be modified using any of the above index notation:

In [71]:
```
x2[0, 0] = 12
x2
```
Out[71]:
```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated. Try it out for yourself!

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (`:`) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array `x`, use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0`, `stop=` *`size of dimension`*, `step=1`. We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

One-dimensional subarrays

In [72]:
```
x = np.arange(10)
x
```
Out[72]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [73]:
```
x[5:]  # elements after index 5
```
Out[73]:
```
array([5, 6, 7, 8, 9])
```

In [74]:
```
x[4:7]  # middle sub-array
```
Out[74]:
```
array([4, 5, 6])
```

In [75]:
```
x[::2]  # every other element
```
Out[75]:
```
array([0, 2, 4, 6, 8])
```

A potentially confusing case is when the `step` value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

In [76]:
```
x[5::-2]  # reversed every other from index 5
```
Out[76]:
```
array([5, 3, 1])
```

Multi-dimensional subarrays

Multi-dimensional slices work in the same way, with multiple slices separated by commas. For example:

In [77]:
```
x2
```
Out[77]:
```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

In [78]:
```
x2[:2, :3]  # two rows, three columns
```
Out[78]:
```
array([[12,  5,  2],
       [ 7,  6,  8]])
```

One commonly needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (`:`):

In [79]:
```
print(x2[:, 1])  # second column of x2
```
```
[5 6 6]
```

Creating copies of arrays

It is sometimes useful to explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

In [80]:
```
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
```
```
[[12  5]
 [ 7  6]]
```

If we now modify this subarray, the original array is not touched:

In [81]:
```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
```
```
[[42  5]
 [ 7  6]]
```

Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape` method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

In [82]:
```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```
```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Note that for this to work, the size of the initial array must match the size of the reshaped array. Where possible, the `reshape` method will use a no-copy view of the initial array, but with non-contiguous memory buffers this is not always the case.

Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. This can be done with the `reshape` method, or more easily done by making use of the `newaxis` keyword within a slice operation:

In [83]:
```
x = np.array([1, 2, 3])

# column vector via reshape
x.reshape((3, 1))
```
Out[83]:
```
array([[1],
       [2],
       [3]])
```

Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

In [84]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```
Out[84]:
```
array([1, 2, 3, 3, 2, 1])
```

It can also be used for two-dimensional arrays:

In [85]:
```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
```

In [86]:
```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```
Out[86]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

In [87]:
```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```
```
[1 2 3] [99 99] [3 2 1]
```

Computation on NumPy Arrays: Universal Functions

Up until now, we have been discussing some of the basic nuts and bolts of NumPy; in the next few sections, we will dive into the reasons that NumPy is so important in the Python data science world. Namely, it provides an easy and flexible interface to optimized computation with arrays of data.

Computation on NumPy arrays can be very fast, or it can be very slow. The key to making it fast is to use *vectorized* operations, generally implemented through NumPy's *universal functions* (ufuncs). This section motivates the need for NumPy's ufuncs, which can be used to make repeated calculations on array elements much more efficient. It then introduces many of the most common and useful arithmetic ufuncs available in the NumPy package.

The Slowness of Loops

Python's default implementation (known as CPython) does some operations very slowly. This is in part due to the dynamic, interpreted nature of the language: the fact that types are flexible, so that sequences of operations cannot be compiled down to efficient machine code as in languages like C and Fortran. Recently there have been various attempts to address this weakness: well-known examples are the PyPy project, a just-in-time compiled implementation of Python; the Cython project, which converts Python code to compilable C code; and the Numba project, which converts snippets of Python code to fast LLVM bytecode. Each of these has its strengths and weaknesses, but it is safe to say that none of the three approaches has yet surpassed the reach and popularity of the standard CPython engine.

The relative sluggishness of Python generally manifests itself in situations where many small operations are being repeated – for instance looping over arrays to operate on each element. For example, imagine we have an array of values and we'd like to compute the reciprocal of each. A straightforward approach might look like this:

In [88]:
```
np.random.seed(0)  # seed for reproducibility

def compute_reciprocals(values):
    output = np.empty(len(values))  # create an empty array of same length as input array
    for i in range(len(values)):
        output[i] = 1.0 / values[i]  # perform reciprocal operation within for loop
    return output

values = np.random.randint(1, 10, size=5)  # 1D array of length 5 with values between 1 to 10
compute_reciprocals(values)
```
Out[88]:
```
array([0.16666667, 1.        , 0.25      , 0.25      , 0.125     ])
```

This implementation probably feels fairly natural to someone from, say, a C or Java background. But if we measure the execution time of this code for a large input, we see that this operation is very slow, perhaps surprisingly so! We'll benchmark this with IPython's `%timeit`

In [89]:
```
big_array = np.random.randint(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
```
```
2.98 s ± 265 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

It takes several seconds to compute these million operations and to store the result! When even cell phones have processing speeds measured in Giga-FLOPS (i.e., billions of numerical operations per second), this seems almost absurdly slow. It turns out that the bottleneck here is not the operations themselves, but the type-checking and function dispatches that CPython must do at each cycle of the loop. Each time the reciprocal is computed, Python first examines the object's type and does a dynamic lookup of the correct function to use for that type. If we were working in compiled code instead, this type specification would be known before the code executes and the result could be computed much more efficiently.

Introducing UFuncs

For many types of operations, NumPy provides a convenient interface into just this kind of statically typed, compiled routine. This is known as a *vectorized* operation. This can be accomplished by simply performing an operation on the array, which will then be applied to each element. This vectorized approach is designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution.

Looking at the execution time for our big array, we see that it completes orders of magnitude faster than the Python loop:

In [90]:
```
%timeit (1.0 / big_array)
```
```
4.92 ms ± 246 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Vectorized operations in NumPy are implemented via *ufuncs*, whose main purpose is to quickly execute repeated operations on values in NumPy arrays. Ufuncs are extremely flexible – operations are not limited to one-dimensional arrays–they can also act on multi-dimensional arrays as well:

In [91]:
```
x = np.arange(9).reshape((3, 3))
2 ** x
```
Out[91]:
```
array([[  1,   2,   4],
       [  8,  16,  32],
       [ 64, 128, 256]], dtype=int32)
```

Computations using vectorization through ufuncs are nearly always more efficient than their counterpart implemented using Python loops, especially as the arrays grow in size. Any time you see such a loop in a Python script, you should consider whether it can be replaced with a vectorized expression.

Exploring NumPy's UFuncs

Ufuncs exist in two flavors: *unary ufuncs*, which operate on a single input, and *binary ufuncs*, which operate on two inputs. We'll see examples of both these types of functions here.

Array arithmetic

NumPy's ufuncs feel very natural to use because they make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

In [92]:
```
x = np.arange(4)
print("x     =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2)  # floor division
print("-x     = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2  = ", x % 2)
```
```
x     = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [0.  0.5 1.  1.5]
x // 2 = [0 0 1 1]
-x     =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2  =  [0 1 0 1]
```

In addition, these can be strung together however you wish, and the standard order of operations is respected:

In [93]:
```
-(0.5*x + 1) ** 2
```
Out[93]:
```
array([-1.  , -2.25, -4.  , -6.25])
```

Advanced Ufunc Features

Many NumPy users make use of ufuncs without ever learning their full set of features. We'll outline a few specialized features of ufuncs here.

Specifying output

For large calculations, it is sometimes useful to be able to specify the array where the result of the calculation will be stored. Rather than creating a temporary array, this can be used to write computation results directly to the memory location where you'd like them to be. For all ufuncs, this can be done using the `out` argument of the function:

In [94]:
```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
```
```
[ 0. 10. 20. 30. 40.]
```

Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed directly from the object. For example, if we'd like to *reduce* an array with a particular operation, we can use the `reduce` method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

In [95]:
```
x = np.arange(1, 6)
np.add.reduce(x)
```
Out[95]:
```
15
```

More information on universal functions (including the full list of available functions) can be found on the NumPy documentation website.

Aggregations: Min, Max, and Everything In Between

Often when faced with a large amount of data, a first step is to compute summary statistics for the data in question. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the "typical" values in a dataset, but other aggregates are useful as well (the sum, product, median, minimum and maximum, quantiles, etc.).

NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and demonstrate some of them here.

Summing the Values in an Array

Consider computing the sum of all values in an array. Python itself can do this using the built-in `sum` function. However, NumPy's version `np.sum` of the operation is computed much more quickly because it executes the operation in compiled code.

In [96]:
```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```
```
215 ms ± 17.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.01 ms ± 131 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Be careful, though: the `sum` function and the `np.sum` function are not identical, which can sometimes lead to confusion! In particular, their optional arguments have different meanings, and `np.sum` is aware of multiple array dimensions, as we will see in the following section.

Minimum and Maximum

Similarly, Python has built-in `min` and `max` functions, used to find the minimum value and maximum value of any given array:

In [97]:
```
min(big_array), max(big_array)
```
Out[97]:
```
(7.071203171893359e-07, 0.9999997207656334)
```

NumPy's corresponding functions have similar syntax, and again operate much more quickly:

In [98]:
```
np.min(big_array), np.max(big_array)
```
Out[98]:
```
(7.071203171893359e-07, 0.9999997207656334)
```

In [99]:
```
%timeit min(big_array)
%timeit np.min(big_array)
```
```
126 ms ± 7.52 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
479 µs ± 68.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Multi dimensional aggregates

One common type of aggregation operation is an aggregate along a row or column. Say you have some data stored in a two-dimensional array:

In [100]:
```
M = np.random.random((3, 4))
print(M)
```
```
[[0.71125897 0.16526245 0.43008959 0.61228252]
 [0.37038403 0.06785738 0.48412739 0.7476983 ]
 [0.47720923 0.23956676 0.87236336 0.36038331]]
```

Aggregation functions take an additional argument specifying the *axis* along which the aggregate is computed. For example, we can find the minimum value within each column by specifying `axis=0`:

In [101]:
```
M.min(axis=0)
```
Out[101]:
```
array([0.37038403, 0.06785738, 0.43008959, 0.36038331])
```

The function returns four values, corresponding to the four columns of numbers. Similarly, we can find the maximum value within each row. The way the axis is specified here can be confusing to users coming from other languages. The `axis` keyword specifies the *dimension of the array that will be collapsed*, rather than the dimension that will be returned. So specifying `axis=0` means that the first axis will be collapsed: for two-dimensional arrays, this means that values within each column will be aggregated.

Other aggregation functions

NumPy provides many other aggregation functions, but we won't discuss them in detail here. Additionally, most aggregates have a `NaN`-safe counterpart that computes the result while ignoring missing/null values, which are marked by the special IEEE floating-point `NaN` value. Some of these `NaN`-safe functions were not added until NumPy 1.8, so they will not be available in older NumPy versions.

The following table provides a list of useful aggregation functions available in NumPy:

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

On to Pandas

There is defintely more to NumPy, it provides several other useful features like array broadcasting, boolean masking, advanced indexing and sorting to name a few. But this is a good time to stop and practice all the core concepts discussed here. As you get more comfortable with these basic building blocks, it will be easier to learn the rest of the features offered by Numpy. Let us now move on to Pandas in the next notebook, which builds on the data structures that we learned in Numpy.