This notebook was adapted from the Python Data Science Handbook by Jake VanderPlas. The content is available on GitHub (https://github.com/jakevdp/PythonDataScienceHandbook).

Introduction to NumPy

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.

If you installed Anaconda, you already have NumPy installed and ready to go, or you can go to http://www.numpy.org/ (http://www.numpy.org/) and follow the installation instructions found there. Once you do, you can import NumPy and double-check the version:

In [1]:

```
import numpy
numpy.__version__
```

Out[1]:

```
'1.16.3'
```

By convention, you'll find that most people will import NumPy using np as an alias:

In [2]:

```
import numpy as np
```

Creating Arrays from Python Lists

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

In [3]:

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

Out[3]:

```
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 [4]:

```
np.array([3.14, 4, 2, 3])
```

Out[4]:

```
array([3.14, 4.  , 2.  , 3.  ])
```

We can use the `dtype` keyword to explicitly set the data type of the resulting array:

In [5]:

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

Out[5]:

```
array([1., 2., 3., 4.], dtype=float32)
```

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 [6]:

```
# nested lists result in multi-dimensional arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])
```

Out[6]:

```
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 [7]:

```
# Create a length-10 integer array filled with zeros
np.zeros(10, dtype=int)
```

Out[7]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [8]:

```
# Create a 3x5 floating-point array filled with ones
np.ones((3, 5), dtype=float)
```

Out[8]:

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

In [9]:

```
# Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)
```

Out[9]:

```
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 [10]:

```
# 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[10]:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [11]:

```
# Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
```

Out[11]:

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

In [12]:

```
# Create a 3x3 array of uniformly distributed random values between 0 and 1
np.random.random((3, 3))
```

Out[12]:

```
array([[0.3475436 , 0.69230766, 0.09504837],
       [0.06984418, 0.12399333, 0.45535407],
       [0.64366291, 0.28957586, 0.37238884]])
```

In [13]:

```
# Create a 3x3 array of normally distributed random values with mean 0 and standard deviati
np.random.normal(0, 1, (3, 3))
```

Out[13]:

```
array([[-1.2055274 ,  1.08906024,  0.71077729],
       [ 1.63297852,  0.45016251, -0.43713546],
       [-0.42372075, -1.04664497,  0.65812464]])
```

In [14]:

```
# Create a 3x3 array of random integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
```

Out[14]:

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

In [15]:

```
# Create a 3x3 identity matrix
np.eye(3)
```

Out[15]:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [16]:

```
# Create an uninitialized array of three integers
# The values will be whatever happens to already exist at that memory location
np.empty(3)
```

Out[16]:

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

NumPy Array Attributes

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 [17]:

```
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):

In [18]:

```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Another useful attribute is the `dtype`, the data type of the array.

In [19]:

```
print("dtype:", x3.dtype)
```

```
dtype: int32
```

Other attributes include `itemsize`, which lists the size (in bytes) of each array element, and `nbytes`, which lists the total size (in bytes) of the array:

In [20]:

```
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
```

```
itemsize: 4 bytes
nbytes: 240 bytes
```

In general, we expect that `nbytes` is equal to `itemsize` times `size`.

Array Indexing: Accessing Single Elements

Indexing in NumPy is similar to Python's standard list indexing. 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 [21]:

```
x1
```

Out[21]:

```
array([5, 0, 3, 3, 7, 9])
```

In [22]:

```
x1[0]
```

Out[22]:

```
5
```

In [23]:

```
x1[4]
```

Out[23]:

```
7
```

To index from the end of the array, you can use negative indices:

In [24]:

```
x1[-1]
```

Out[24]:

```
9
```

In [25]:

```
x1[-2]
```

Out[25]:

```
7
```

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

In [26]:

```
x2
```

Out[26]:

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

In [27]:

```
x2[0, 0]
```

Out[27]:

```
3
```

In [28]:

```
x2[2, 0]
```

Out[28]:

```
1
```

In [29]:

```
x2[2, -1]
```

Out[29]:

```
7
```

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

In [30]:

```
x2[0, 0] = 12
x2
```

Out[30]:

```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

Remember 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.

In [31]:

```
x1[0] = 3.14159  # this will be truncated!
x1
```

Out[31]:

```
array([3, 0, 3, 3, 7, 9])
```

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 [32]:

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

Out[32]:

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

In [33]:

```
x[:5]  # first five elements
```

Out[33]:

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

In [34]:

```
x[5:]  # elements after index 5
```

Out[34]:

```
array([5, 6, 7, 8, 9])
```

In [35]:

```
x[4:7]  # middle sub-array
```

Out[35]:

```
array([4, 5, 6])
```

In [36]:

```
x[::2]  # every other element
```

Out[36]:

```
array([0, 2, 4, 6, 8])
```

In [37]:

```
x[1::2]  # every other element, starting at index 1
```

Out[37]:

```
array([1, 3, 5, 7, 9])
```

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 [38]:

```
x[::-1]  # all elements, reversed
```

Out[38]:

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

In [39]:

```
x[5::-2]  # reversed every other from index 5
```

Out[39]:

```
array([5, 3, 1])
```

Multi-dimensional subarrays

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

In [40]:

```
x2
```

Out[40]:

```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

In [41]:

```
x2[:2, :3]  # two rows, three columns
```

Out[41]:

```
array([[12,  5,  2],
       [ 7,  6,  8]])
```

In [42]:

```
x2[:3, ::2]  # all rows, every other column
```

Out[42]:

```
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])
```

Finally, subarray dimensions can even be reversed together:

In [43]:

```
x2[::-1, ::-1]
```

Out[43]:

```
array([[ 7,  7,  6,  1],
       [ 8,  8,  6,  7],
       [ 4,  2,  5, 12]])
```

Accessing array rows and columns

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 [44]:

```
print(x2[:, 0])  # first column of x2
```

```
[12  7  1]
```

In [45]:

```
print(x2[0, :])  # first row of x2
```

```
[12  5  2  4]
```

In the case of row access, the empty slice can be omitted for a more compact syntax:

In [46]:

```
print(x2[0])  # equivalent to x2[0, :]
```

```
[12  5  2  4]
```

Subarrays as no-copy views

One important–and extremely useful–thing to know about array slices is that they return *views* rather than *copies* of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Consider our two-dimensional array from before:

In [47]:

```
print(x2)
```

```
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Let's extract a 2×2 subarray from this:

In [48]:

```
x2_sub = x2[:2, :2]
print(x2_sub)
```

```
[[12  5]
 [ 7  6]]
```

If we modify this subarray, we'll see that the original array is changed:

In [49]:

```
x2_sub[0, 0] = 99
print(x2_sub)
```

```
[[99  5]
 [ 7  6]]
```

In [50]:

```
print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

This default behavior is actually quite useful: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

Creating copies of arrays

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

In [51]:

```
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
```

```
[[99  5]
 [ 7  6]]
```

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

In [52]:

```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
```

In [53]:

```
print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

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 [54]:

```
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 [55]:

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

# row vector via reshape
x.reshape((1, 3))
```

Out[55]:

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

In [56]:

```
# row vector via newaxis
x[np.newaxis, :]
```

Out[56]:

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

In [57]:

```
# column vector via reshape
x.reshape((3, 1))
```

Out[57]:

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

In [58]:

```
# column vector via newaxis
x[:, np.newaxis]
```

Out[58]:

```
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 [59]:

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

Out[59]:

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

You can also concatenate more than two arrays at once:

In [60]:

```
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

It can also be used for two-dimensional arrays:

In [61]:

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

In [62]:

```
# concatenate along the first axis
np.concatenate([grid, grid])
```

Out[62]:

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

In [63]:

```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

Out[63]:

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

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions:

In [64]:

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

# vertically stack the arrays
np.vstack([x, grid])
```

Out[64]:

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

In [65]:

```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

Out[65]:

```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

Similary, `np.dstack` will stack arrays along the third axis.

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 [66]:

```
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]
```

Notice that *N* split-points, leads to *N + 1* subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

In [67]:

```
grid = np.arange(16).reshape((4, 4))
grid
```

Out[67]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [68]:

```
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

In [69]:

```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Similarly, `np.dsplit` will split arrays along the third axis.

Computation on NumPy Arrays

Array arithmetic

In [70]:

```
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
```

```
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]
```

Negation, and a `**` operator for exponentiation, and a `%` operator for modulus:

In [71]:

```
print("-x     = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2  = ", x % 2)
```

```
-x     =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2  =  [0 1 0 1]
```

In [72]:

```
-(0.5*x + 1) ** 2
```

Out[72]:

```
array([-1.  , -2.25, -4.  , -6.25])
```

Each of these arithmetic operations are simply convenient wrappers around specific functions built into NumPy; for example, the `+` operator is a wrapper for the `add` function:

In [73]:

```
np.add(x, 2)
```

Out[73]:

```
array([2, 3, 4, 5])
```

The following table lists the arithmetic operators implemented in NumPy:

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Absolute value

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's built-in absolute value function:

In [74]:

```
x = np.array([-2, -1, 0, 1, 2])
abs(x)
```

Out[74]:

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

The corresponding NumPy function is `np.absolute`, which is also available under the alias `np.abs`:

In [75]:

```
np.absolute(x)
```

Out[75]:

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

In [76]:

```
np.abs(x)
```

Out[76]:

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

Trigonometric functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We'll start by defining an array of angles:

In [77]:

```
theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

In [78]:

```
print("theta      = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
```

```
theta      =  [0.         1.57079633 3.14159265]
sin(theta) =  [0.0000000e+00 1.0000000e+00 1.2246468e-16]
cos(theta) =  [ 1.000000e+00  6.123234e-17 -1.000000e+00]
tan(theta) =  [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

The values are computed to within machine precision, which is why values that should be zero do not always hit exactly zero. Inverse trigonometric functions are also available:

In [79]:

```
x = [-1, 0, 1]
print("x         = ", x)
print("arcsin(x) = ", np.arcsin(x))
print("arccos(x) = ", np.arccos(x))
print("arctan(x) = ", np.arctan(x))
```

```
x         =  [-1, 0, 1]
arcsin(x) =  [-1.57079633  0.          1.57079633]
arccos(x) =  [3.14159265 1.57079633 0.        ]
arctan(x) =  [-0.78539816  0.          0.78539816]
```

Exponents and logarithms

Another common type of operation available in a NumPy ufunc are the exponentials:

In [80]:

```
x = [1, 2, 3]
print("x     =", x)
print("e^x   =", np.exp(x))
print("2^x   =", np.exp2(x))
print("3^x   =", np.power(3, x))
```

```
x     = [1, 2, 3]
e^x   = [ 2.71828183  7.3890561  20.08553692]
2^x   = [2. 4. 8.]
3^x   = [ 3  9 27]
```

The inverse of the exponentials, the logarithms, are also available. The basic `np.log` gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well:

In [81]:

```
x = [1, 2, 4, 10]
print("x        =", x)
print("ln(x)    =", np.log(x))
print("log2(x)  =", np.log2(x))
print("log10(x) =", np.log10(x))
```

```
x        = [1, 2, 4, 10]
ln(x)    = [0.         0.69314718 1.38629436 2.30258509]
log2(x)  = [0.         1.         2.         3.32192809]
log10(x) = [0.         0.30103    0.60205999 1.        ]
```

There are also some specialized versions that are useful for maintaining precision with very small input:

In [82]:

```
x = [0, 0.001, 0.01, 0.1]
print("exp(x) - 1 =", np.expm1(x))
print("log(1 + x) =", np.log1p(x))
```

```
exp(x) - 1 = [0.         0.0010005  0.01005017 0.10517092]
log(1 + x) = [0.         0.0009995  0.00995033 0.09531018]
```

When `x` is very small, these functions give more precise values than if the raw `np.log` or `np.exp` were to be used.

Advanced Features

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. This can be done using the `out` argument of the function:

In [83]:

```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
```

```
[ 0. 10. 20. 30. 40.]
```

This can even be used with array views. For example, we can write the results of a computation to every other element of a specified array:

In [84]:

```
y = np.zeros(10)
np.power(2, x, out=y[::2])
print(y)
```

```
[ 1.  0.  2.  0.  4.  0.  8.  0. 16.  0.]
```

If we had instead written `y[::2] = 2 ** x`, this would have resulted in the creation of a temporary array to hold the results of `2 ** x`, followed by a second operation copying those values into the `y` array. This doesn't make much of a difference for such a small computation, but for very large arrays the memory savings from careful use of the `out` argument can be significant.

Aggregates

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. 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` function returns the sum of all elements in the array:

In [85]:

```
x = np.arange(1, 6)
np.add.reduce(x)
```

Out[85]:

Similarly, calling `reduce` on the `multiply` ufunc results in the product of all array elements:

In [86]:

```
np.multiply.reduce(x)
```

Out[86]:

```
120
```

If we'd like to store all the intermediate results of the computation, we can instead use `accumulate`:

In [87]:

```
np.add.accumulate(x)
```

Out[87]:

```
array([ 1,  3,  6, 10, 15], dtype=int32)
```

In [88]:

```
np.multiply.accumulate(x)
```

Out[88]:

```
array([  1,   2,   6,  24, 120], dtype=int32)
```

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`).

Outer products

In [89]:

```
x = np.arange(1, 6)
np.multiply.outer(x, x)
```

Out[89]:

```
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

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

As a quick example, consider computing the sum of all values in an array. Python itself can do this using the built-in `sum` function:

In [90]:

```
import numpy as np
```

In [91]:

```
L = np.random.random(100)
sum(L)
```

Out[91]:

```
52.12818058833704
```

The syntax is quite similar to that of NumPy's `sum` function, and the result is the same in the simplest case:

In [92]:

```
np.sum(L)
```

Out[92]:

```
52.12818058833702
```

However, NumPy's version of the operation is computed much more quickly:

In [93]:

```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```

```
60.4 ms ± 457 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
615 µs ± 46.4 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 [94]:

```
min(big_array), max(big_array)
```

Out[94]:

```
(1.4057692298008462e-06, 0.9999994392723005)
```

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

In [95]:

```
np.min(big_array), np.max(big_array)
```

Out[95]:

```
(1.4057692298008462e-06, 0.9999994392723005)
```

In [96]:

```
%timeit min(big_array)
%timeit np.min(big_array)
```

```
42.6 ms ± 329 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
296 µs ± 12.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

For `min`, `max`, `sum`, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself:

In [97]:

```
print(big_array.min(), big_array.max(), big_array.sum())
```

```
1.4057692298008462e-06 0.9999994392723005 500202.5348847683
```

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 [98]:

```
M = np.random.random((3, 4))
print(M)
```

```
[[0.50063048 0.07383653 0.49018646 0.72521956]
 [0.84926562 0.10226215 0.99559424 0.59250301]
 [0.53509    0.88518089 0.25518136 0.13130483]]
```

By default, each NumPy aggregation function will return the aggregate over the entire array:

In [99]:

```
M.sum()
```

Out[99]:

```
6.1362551272647154
```

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 [100]:

```
M.min(axis=0)
```

Out[100]:

```
array([0.50063048, 0.07383653, 0.25518136, 0.13130483])
```

The function returns four values, corresponding to the four columns of numbers.

Similarly, we can find the maximum value within each row:

In [101]:

```
M.max(axis=1)
```

Out[101]:

```
array([0.72521956, 0.99559424, 0.88518089])
```

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 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 median 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

Function Name	NaN-safe Version	Description
`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

Computation on Arrays: Broadcasting

We have seen how NumPy's universal functions can be used to *vectorize* operations and thereby remove slow Python loops. Another means of vectorizing operations is to use NumPy's *broadcasting* functionality. Broadcasting is simply a set of rules for applying binary ufuncs (e.g., addition, subtraction, multiplication, etc.) on arrays of different sizes.

Introducing Broadcasting

Recall that for arrays of the same size, binary operations are performed on an element-by-element basis:

In [102]:

```
import numpy as np
```

In [103]:

```
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

Out[103]:

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

Broadcasting allows these types of binary operations to be performed on arrays of different sizes–for example, we can just as easily add a scalar (think of it as a zero-dimensional array) to an array:

In [104]:

```
a + 5
```

Out[104]:

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

We can think of this as an operation that stretches or duplicates the value `5` into the array `[5, 5, 5]`, and adds the results. The advantage of NumPy's broadcasting is that this duplication of values does not actually take place, but it is a useful mental model as we think about broadcasting.

We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array:

In [105]:

```
M = np.ones((3, 3))
M
```

Out[105]:

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

In [106]:

```
M + a
```

Out[106]:

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

Here the one-dimensional array `a` is stretched, or broadcast across the second dimension in order to match the shape of `M`.

While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider the following example:

In [107]:

```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]

print(a)
print(b)
```

```
[0 1 2]
[[0]
 [1]
 [2]]
```

In [108]:

```
a + b
```

Out[108]:

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

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
- Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

To make these rules clear, let's consider a few examples in detail.

Broadcasting example 1

Let's look at adding a two-dimensional array to a one-dimensional array:

In [109]:

```
M = np.ones((2, 3))
a = np.arange(3)
```

Let's consider an operation on these two arrays. The shape of the arrays are

- `M.shape = (2, 3)`
- `a.shape = (3,)`

We see by rule 1 that the array `a` has fewer dimensions, so we pad it on the left with ones:

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- `M.shape -> (2, 3)`
- `a.shape -> (2, 3)`

The shapes match, and we see that the final shape will be `(2, 3)`:

In [110]:

```
M + a
```

Out[110]:

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

Broadcasting example 2

Let's take a look at an example where both arrays need to be broadcast:

In [111]:

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

Again, we'll start by writing out the shape of the arrays:

- `a.shape = (3, 1)`
- `b.shape = (3,)`

Rule 1 says we must pad the shape of `b` with ones:

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

- `a.shape -> (3, 3)`
- `b.shape -> (3, 3)`

Because the result matches, these shapes are compatible. We can see this here:

In [112]:

```
a + b
```

Out[112]:

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

Broadcasting example 3

Now let's take a look at an example in which the two arrays are not compatible:

In [113]:

```
M = np.ones((3, 2))
a = np.arange(3)
```

This is just a slightly different situation than in the first example: the matrix `M` is transposed. How does this affect the calculation? The shape of the arrays are

- `M.shape = (3, 2)`
- `a.shape = (3,)`

Again, rule 1 tells us that we must pad the shape of `a` with ones:

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

By rule 2, the first dimension of `a` is stretched to match that of `M`:

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

Now we hit rule 3–the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

In [114]:

```
M + a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-114-8cac1d547906> in <module>()
----> 1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Note the potential confusion here: you could imagine making `a` and `M` compatible by, say, padding `a`'s shape with ones on the right rather than the left. But this is not how the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side padding is what you'd like, you can do this explicitly by reshaping the array.

In [115]:

```
a[:, np.newaxis].shape
```

Out[115]:

```
(3, 1)
```

In [116]:

```
M + a[:, np.newaxis]
```

Out[116]:

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

Also note that while we've been focusing on the `+` operator here, these broadcasting rules apply to *any* binary `ufunc`. For example, here is the `logaddexp(a, b)` function, which computes `log(exp(a) + exp(b))` with more precision than the naive approach:

In [117]:

```
np.logaddexp(M, a[:, np.newaxis])
```

Out[117]:

```
array([[1.31326169, 1.31326169],
       [1.69314718, 1.69314718],
       [2.31326169, 2.31326169]])
```

Broadcasting in Practice

Centering an array

Imagine you have an array of 10 observations, each of which consists of 3 values. Using the standard convention, we'll store this in a 10×3 array:

In [118]:

```
X = np.random.random((10, 3))
```

We can compute the mean of each feature using the `mean` aggregate across the first dimension:

In [119]:

```
Xmean = X.mean(0)
Xmean
```

Out[119]:

```
array([0.48773885, 0.42332575, 0.50590496])
```

And now we can center the `X` array by subtracting the mean (this is a broadcasting operation):

In [120]:

```
X_centered = X - Xmean
```

To double-check that we've done this correctly, we can check that the centered array has near zero mean:

In [121]:

```
X_centered.mean(0)
```

Out[121]:

```
array([ 5.55111512e-18, -4.44089210e-17,  8.88178420e-17])
```

To within machine precision, the mean is now zero.

Plotting a two-dimensional function

One place that broadcasting is very useful is in displaying images based on two-dimensional functions. If we want to define a function $z = f(x, y)$, broadcasting can be used to compute the function across the grid:

In [122]:

```
# x and y have 50 steps from 0 to 5
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[:, np.newaxis]

z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

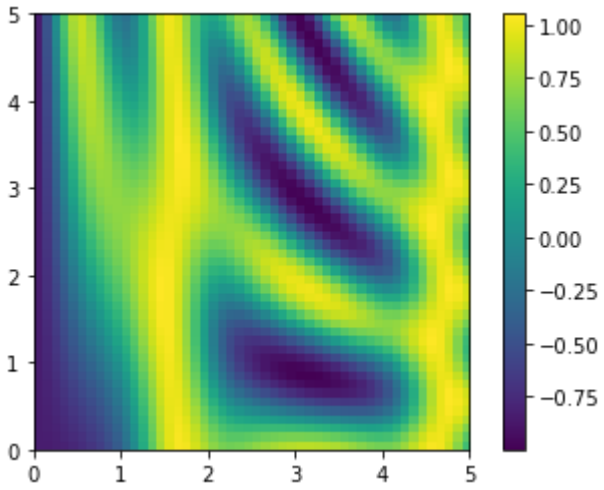
We'll use Matplotlib to plot this two-dimensional array. Matplotlib will be discussed later.

In [123]:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

In [124]:

```
plt.imshow(z, origin='lower', extent=[0, 5, 0, 5], cmap='viridis')
plt.colorbar();
```



The result is a compelling visualization of the two-dimensional function.

Comparisons, Masks, and Boolean Logic

Comparison Operators

We have discussed some arithmetic operators. We saw that using `+`, `-`, `*`, `/`, and others on arrays leads to element-wise operations. NumPy also implements comparison operators such as `<` (less than) and `>` (greater than) as element-wise ufuncs. The result of these comparison operators is always an array with a Boolean data type. All six of the standard comparison operations are available:

In [125]:

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

In [126]:

```
x < 3  # less than
```

Out[126]:

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

In [127]:

```
x > 3  # greater than
```

Out[127]:

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

In [128]:

```
x <= 3  # less than or equal
```

Out[128]:

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

In [129]:

```
x >= 3  # greater than or equal
```

Out[129]:

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

In [130]:

```
x != 3  # not equal
```

Out[130]:

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

In [131]:

```
x == 3  # equal
```

Out[131]:

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

It is also possible to do an element-wise comparison of two arrays, and to include compound expressions:

In [132]:

```
(2 * x) == (x ** 2)
```

Out[132]:

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

As in the case of arithmetic operators, the comparison operators are implemented as ufuncs in NumPy; for example, when you write `x < 3`, internally NumPy uses `np.less(x, 3)`. A summary of the comparison operators and their equivalent ufunc is shown here:

Operator	Equivalent ufunc	Operator	Equivalent ufunc
==	np.equal	!=	np.not_equal
<	np.less	<=	np.less_equal
>	np.greater	>=	np.greater_equal

These will work on arrays of any size and shape. Here is a two-dimensional example:

In [133]:

```
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
x
```

Out[133]:

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

In [134]:

```
x < 6
```

Out[134]:

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

In each case, the result is a Boolean array, and NumPy provides a number of straightforward patterns for working with these Boolean results.

Working with Boolean Arrays

Given a Boolean array, there are a host of useful operations you can do. We'll work with `x`, the two-dimensional array we created earlier.

In [135]:

```
print(x)
```

```
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
```

Counting entries

To count the number of `True` entries in a Boolean array, `np.count_nonzero` is useful:

In [136]:

```
# how many values less than 6?
np.count_nonzero(x < 6)
```

Out[136]:

```
8
```

We see that there are eight array entries that are less than 6. Another way to get at this information is to use `np.sum`; in this case, `False` is interpreted as `0`, and `True` is interpreted as `1`:

In [137]:

```
np.sum(x < 6)
```

Out[137]:

```
8
```

The benefit of `sum()` is that like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

In [138]:

```
# how many values less than 6 in each row?
np.sum(x < 6, axis=1)
```

Out[138]:

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

This counts the number of values less than 6 in each row of the matrix.

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) `np.any` or `np.all`:

In [139]:

```
# are there any values greater than 8?
np.any(x > 8)
```

Out[139]:

```
True
```

In [140]:

```
# are there any values less than zero?
np.any(x < 0)
```

Out[140]:

```
False
```

In [141]:

```
# are all values less than 10?
np.all(x < 10)
```

Out[141]:

```
True
```

In [142]:

```
# are all values equal to 6?
np.all(x == 6)
```

Out[142]:

```
False
```

`np.all` and `np.any` can be used along particular axes as well. For example:

In [143]:

```
# are all values in each row less than 4?
np.all(x < 8, axis=1)
```

Out[143]:

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

Here all the elements in the first and third rows are less than 8, while this is not the case for the second row.

Finally, a quick warning: as mentioned before, Python has built-in `sum()`, `any()`, and `all()` functions. These have a different syntax than the NumPy versions, and in particular will fail or produce unintended results when used on multidimensional arrays. Be sure that you are using `np.sum()`, `np.any()`, and `np.all()` for these examples!

Boolean operators

Like with the standard arithmetic operators, NumPy overloads *bitwise logic operators*, `&`, `|`, `^`, and `~`, as ufuncs which work element-wise on (usually Boolean) arrays.

For example, we can address this sort of compound question as follows:

In [144]:

```
x
```

Out[144]:

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

In [145]:

```
np.sum((x > 3) & (x < 7))
```

Out[145]:

```
4
```

Boolean Arrays as Masks

Another powerful pattern is to use Boolean arrays as masks, to select particular subsets of the data themselves. Returning to our `x` array from before, suppose we want an array of all values in the array that are less than, say, 5:

In [146]:

```
x
```

Out[146]:

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

We can obtain a Boolean array for this condition easily, as we've already seen:

In [147]:

```
x < 5
```

Out[147]:

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

Now to *select* these values from the array, we can simply index on this Boolean array; this is known as a *masking* operation:

In [148]:

```
x[x < 5]
```

Out[148]:

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

What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is `True`.

Sorting Arrays

In [149]:

```
import numpy as np

def selection_sort(x):
    for i in range(len(x)):
        swap = i + np.argmin(x[i:])
        (x[i], x[swap]) = (x[swap], x[i])
    return x
```

In [150]:

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

Out[150]:

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

In [151]:

```
def bogosort(x):
    while np.any(x[:-1] > x[1:]):
        np.random.shuffle(x)
    return x
```

In [152]:

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

Out[152]:

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

Python contains built-in sorting algorithms that are *much* more efficient than either of the simplistic algorithms just shown. We'll start by looking at the Python built-ins, and then take a look at the routines included in NumPy and optimized for NumPy arrays.

Fast Sorting in NumPy: `np.sort` and `np.argsort`

Although Python has built-in `sort` and `sorted` functions to work with lists, we won't discuss them here because NumPy's `np.sort` function turns out to be much more efficient and useful for our purposes. By default `np.sort` uses an $\mathcal{O}[N \log N]$, *quicksort* algorithm, though *mergesort* and *heapsort* are also available. For most applications, the default quicksort is more than sufficient.

To return a sorted version of the array without modifying the input, you can use `np.sort`:

In [153]:

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

Out[153]:

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

If you prefer to sort the array in-place, you can instead use the `sort` method of arrays:

In [154]:

```
x.sort()
print(x)
```

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

A related function is `argsort`, which instead returns the *indices* of the sorted elements:

In [155]:

```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x)
print(i)
```

```
[1 0 3 2 4]
```

The first element of this result gives the index of the smallest element, the second value gives the index of the second smallest, and so on. These indices can then be used to construct the sorted array if desired:

In [156]:

```
x[i]
```

Out[156]:

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

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the `axis` argument. For example:

In [157]:

```
rand = np.random.RandomState(42)
X = rand.randint(0, 10, (4, 6))
print(X)
```

```
[[6 3 7 4 6 9]
 [2 6 7 4 3 7]
 [7 2 5 4 1 7]
 [5 1 4 0 9 5]]
```

In [158]:

```
# sort each column of X
np.sort(X, axis=0)
```

Out[158]:

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

In [159]:

```
# sort each row of X
np.sort(X, axis=1)
```

Out[159]:

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

Keep in mind that this treats each row or column as an independent array, and any relationships between the row or column values will be lost!

Partial Sorts: Partitioning

Sometimes we're not interested in sorting the entire array, but simply want to find the *k* smallest values in the array. NumPy provides this in the `np.partition` function. `np.partition` takes an array and a number *K*; the result is a new array with the smallest *K* values to the left of the partition, and the remaining values to the right, in arbitrary order:

In [160]:

```
x = np.array([7, 2, 3, 1, 6, 5, 4])
np.partition(x, 3)
```

Out[160]:

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

Note that the first three values in the resulting array are the three smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have arbitrary order.

Similarly to sorting, we can partition along an arbitrary axis of a multidimensional array:

In [161]:

```
np.partition(X, 2, axis=1)
```

Out[161]:

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

The result is an array where the first two slots in each row contain the smallest values from that row, with the remaining values filling the remaining slots.

Finally, just as there is a `np.argsort` that computes indices of the sort, there is a `np.argpartition` that computes indices of the partition.

In [ ]:

```

```