Handout Week 2: NumPy for Arrays

Dr Ian (Yinglong) He

Click to download the Jupyter Notebook files (.ipynb)

based on the notes by Jake VanderPlas

Contents

1 Introduction

This handout outlines techniques for effectively loading, storing, and manipulating in-memory data in Python. The topic is very broad: datasets can come from a wide range of sources and a wide range of formats, including be 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, images–particularly digital images–can be thought of as simply two-dimensional arrays of numbers representing pixel brightness across the area. Sound clips can be thought of as one-dimensional arrays of intensity versus time. 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. We'll now take a look at the specialized tools that Python has for handling such numerical arrays: the NumPy package, and the Pandas package.

This handout 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.

If you followed the advice outlined in the Preface and installed the Anaconda stack, you already have NumPy installed and ready to go. If you're more the do-it-yourself type, you can go to http://www.numpy.org/ and follow the installation instructions found there. Once you do, you can import NumPy and double-check the version:

[1]:
```
import numpy
numpy.__version__
```

[1]:
```
'1.23.3'
```

For the pieces of the package discussed here, I'd recommend NumPy version 1.8 or later. By convention, you'll find that most people in the SciPy/PyData world will import NumPy using `np` as an alias:

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

Throughout this handout, and indeed the rest of the book, you'll find that this is the way we will import and use NumPy.

2 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. While the types of operations shown here may seem a bit dry and pedantic, they comprise the building blocks of many other examples used throughout the book. 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

2.1 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:

[3]:
```
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):

[4]:
```
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
```

2.2 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:

[5]:
```
x1
```

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

[6]:
```
x1[0]
```

[6]:
```
5
```

[7]:
```
x1[4]
```

[7]:
```
7
```

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

[8]:
```
x1[-1]
```

[8]:
```
9
```

[9]:
```
x1[-2]
```

[9]:
```
7
```

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

[10]:
```
x2
```

[10]:
```
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

[11]:
```
x2[0, 0]
```

[11]:
```
3
```

[12]:
```
x2[2, 0]
```

[12]:
```
1
```

[13]:
```
x2[2, -1]
```

[13]:
```
7
```

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

[14]:
```
x2[0, 0] = 12
x2
```

[14]:
```
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. Don't be caught unaware by this behavior!

[15]:
```
x1[0] = 3.14159  # this will be truncated!
x1
```

[15]:
```
array([3, 0, 3, 3, 7, 9])
```

2.3 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.

2.3.1 One-dimensional subarrays

```
[16]: x = np.arange(10)
      x
```

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

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

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

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

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

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

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

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

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

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

```
[21]: 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:

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

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

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

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

2.3.2 Multi-dimensional subarrays

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

```
[24]: x2
```

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

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

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

[26]:
```
x2[:3, ::2]  # all rows, every other column
```

[26]:
```
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])
```

Finally, subarray dimensions can even be reversed together:

[27]:
```
x2[::-1, ::-1]
```

[27]:
```
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 (:):

[28]:
```
print(x2[:, 0])  # first column of x2
```

```
[12  7  1]
```

[29]:
```
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:

[30]:
```
print(x2[0])  # equivalent to x2[0, :]
```

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

2.3.3 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:

[31]:
```
print(x2)
```

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

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

[32]:
```
x2_sub = x2[:2, :2]
print(x2_sub)
```

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

Now if we modify this subarray, we'll see that the original array is changed! Observe:

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

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

```
[34]: 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.

2.3.4 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:

```
[35]: 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:

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

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

```
[37]: print(x2)
```

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

2.4 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:

```
[38]: 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:

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

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

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

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

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

```
[41]: # column vector via reshape
      x.reshape((3, 1))
```

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

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

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

We will see this type of transformation often throughout the remainder of the book.

2.5 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.

2.5.1 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:

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

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

You can also concatenate more than two arrays at once:

```
[44]: 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:

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

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

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

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

```
[47]: 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:

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

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

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

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

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

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

2.5.2 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:

[50]:
```
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:

[51]:
```
grid = np.arange(16).reshape((4, 4))
grid
```

[51]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[52]:
```
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]]
```

[53]:
```
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.

3 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.

3.1 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:

[54]:
```
np.random.seed(0)

def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0 / values[i]
    return output

values = np.random.randint(1, 10, size=5)
compute_reciprocals(values)
```

[54]:
```
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` magic:

[55]:
```
big_array = np.random.randint(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
```

```
1.75 s ± 70.2 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.

3.2 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.

Compare the results of the following two:

```
[56]: print(compute_reciprocals(values))
      print(1.0 / values)
```

```
[0.16666667 1.         0.25       0.25       0.125     ]
[0.16666667 1.         0.25       0.25       0.125     ]
```

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

```
[57]: %timeit (1.0 / big_array)
```

```
3.23 ms ± 193 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 – before we saw an operation between a scalar and an array, but we can also operate between two arrays:

```
[58]: np.arange(5) / np.arange(1, 6)
```

```
[58]: array([0.        , 0.5       , 0.66666667, 0.75      , 0.8       ])
```

And ufunc operations are not limited to one-dimensional arrays–they can also act on multi-dimensional arrays as well:

```
[59]: x = np.arange(9).reshape((3, 3))
      2 ** x
```

```
[59]: 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.

3.3 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.

3.3.1 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:

[60]:
```python
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]
```

There is also a unary ufunc for negation, and a ** operator for exponentiation, and a % operator for modulus:

[61]:
```python
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 addition, these can be strung together however you wish, and the standard order of operations is respected:

[62]:
```python
-(0.5*x + 1) ** 2
```

[62]:
```
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:

[63]:
```python
np.add(x, 2)
```

[63]:
```
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`)

3.3.2 Absolute value

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

[64]:
```
x = np.array([-2, -1, 0, 1, 2])
abs(x)
```

[64]:
```
array([2, 1, 0, 1, 2])
```

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

[65]:
```
np.absolute(x)
```

[65]:
```
array([2, 1, 0, 1, 2])
```

[66]:
```
np.abs(x)
```

[66]:
```
array([2, 1, 0, 1, 2])
```

This ufunc can also handle complex data, in which the absolute value returns the magnitude:

[67]:
```
x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
np.abs(x)
```

[67]:
```
array([5., 5., 2., 1.])
```

3.3.3 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:

[68]:
```
theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

[69]:
```
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:

[70]:
```
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]
```

3.3.4 Exponents and logarithms

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

[71]:
```
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:

[72]:
```
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:

[73]:
```
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.

4 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.

4.1 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:

[74]:
```python
L = np.random.random(100)
sum(L)
```

[74]:
```
50.461758453195614
```

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

[75]:
```python
np.sum(L)
```

[75]:
```
50.46175845319564
```

However, because it executes the operation in compiled code, NumPy's version of the operation is computed much more quickly:

[76]:
```python
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```

```
81 ms ± 3.85 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
924 µs ± 133 µs per loop (mean ± std. dev. of 7 runs, 1,000 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.

4.2 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:

[77]:
```python
min(big_array), max(big_array)
```

[77]:
```
(7.071203171893359e-07, 0.9999997207656334)
```

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

[78]:
```python
np.min(big_array), np.max(big_array)
```

[78]:
```
(7.071203171893359e-07, 0.9999997207656334)
```

[79]:
```python
%timeit min(big_array)
%timeit np.min(big_array)
```

```
57.8 ms ± 3.18 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
360 µs ± 39.4 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

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

[80]:
```
print(big_array.min(), big_array.max(), big_array.sum())
```

```
7.071203171893359e-07 0.9999997207656334 500216.8034810001
```

Whenever possible, make sure that you are using the NumPy version of these aggregates when operating on NumPy arrays!

4.2.1 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:

[81]:
```
M = np.random.random((3, 4))
print(M)
```

```
[[0.79832448 0.44923861 0.95274259 0.03193135]
 [0.18441813 0.71417358 0.76371195 0.11957117]
 [0.37578601 0.11936151 0.37497044 0.22944653]]
```

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

[82]:
```
M.sum()
```

[82]:
```
5.1136763453287335
```

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`:

[83]:
```
M.min(axis=0)
```

[83]:
```
array([0.18441813, 0.11936151, 0.37497044, 0.03193135])
```

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

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

[84]:
```
M.max(axis=1)
```

[84]:
```
array([0.95274259, 0.76371195, 0.37578601])
```

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.

4.2.2 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 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

We will see these aggregates often throughout the rest of the book.

4.3 Example: What is the Average Height of US Presidents?

Aggregates available in NumPy can be extremely useful for summarizing a set of values. As a simple example, let's consider the heights of all US presidents. This data is available in the file *president_heights.csv*, which is a simple comma-separated list of labels and values:

[85]:
```
import pandas as pd
data = pd.read_csv('data/president_heights.csv')
data.head()
```

[85]:
```
   order               name  height(cm)
0      1  George Washington         189
1      2         John Adams         170
2      3   Thomas Jefferson         189
3      4      James Madison         163
4      5       James Monroe         183
```

We'll use the Pandas package, which we'll explore more fully in the following weeks of this course, to read the file and extract this information (note that the heights are measured in centimeters).

[86]:
```
heights = np.array(data['height(cm)'])
print(heights)
```

```
[189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193 178 173
 174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193 182 183
 177 185 188 188 182 185]
```

Now that we have this data array, we can compute a variety of summary statistics:

[87]:
```
print("Mean height:       ", heights.mean())
print("Standard deviation:", heights.std())
print("Minimum height:    ", heights.min())
print("Maximum height:    ", heights.max())
```

```
Mean height:        179.73809523809524
Standard deviation: 6.931843442745892
Minimum height:     163
Maximum height:     193
```

Note that in each case, the aggregation operation reduced the entire array to a single summarizing value, which gives us information about the distribution of values. We may also wish to compute quantiles:

[88]:
```
print("25th percentile:   ", np.percentile(heights, 25))
print("Median:            ", np.median(heights))
print("75th percentile:   ", np.percentile(heights, 75))
```

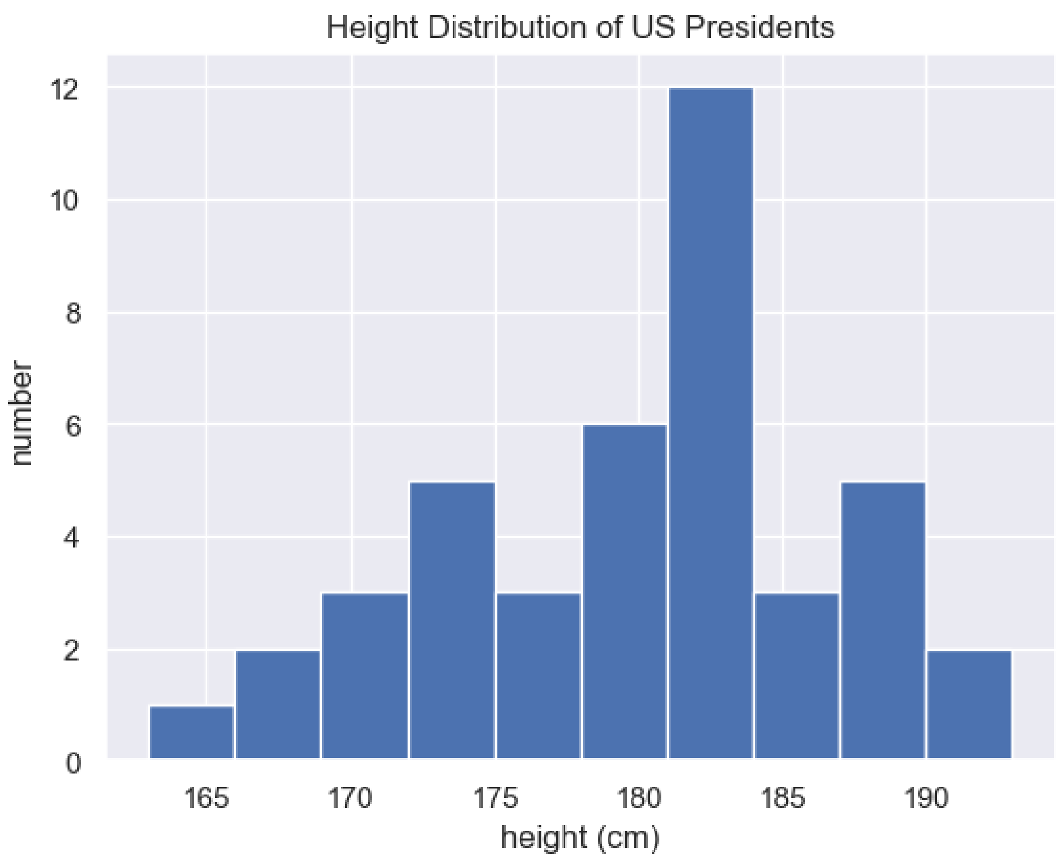
```
25th percentile:    174.25
Median:             182.0
75th percentile:    183.0
```

We see that the median height of US presidents is 182 cm, or just shy of six feet.

Of course, sometimes it's more useful to see a visual representation of this data, which we can accomplish using tools in Matplotlib (we'll discuss Matplotlib more fully in the following weeks of this course). For example, this code generates the following chart:

[89]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()  # set plot style
```

[90]:
```
plt.hist(heights)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number');
```



5 Fancy Indexing

In the previous sections, we saw how to access and modify portions of arrays using simple indices (e.g., `arr[0]`), slices (e.g., `arr[:5]`), and Boolean masks (e.g., `arr[arr > 0]`). In this section, we'll look at another style of array indexing, known as *fancy indexing*. Fancy indexing is like the simple indexing we've already seen, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

5.1 Exploring Fancy Indexing

Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once. For example, consider the following array:

[91]:
```
rand = np.random.RandomState(42)

x = rand.randint(100, size=10)
print(x)
```

```
[51 92 14 71 60 20 82 86 74 74]
```

Suppose we want to access three different elements. We could do it like this:

[92]:
```
[x[3], x[7], x[2]]
```

[92]:
```
[71, 86, 14]
```

Alternatively, we can pass a single list or array of indices to obtain the same result:

[93]:
```
ind = [3, 7, 4]
x[ind]
```

[93]:
```
array([71, 86, 60])
```

When using fancy indexing, the shape of the result reflects the shape of the *index arrays* rather than the shape of the *array being indexed*:

[94]:
```
ind = np.array([[3, 7],
                [4, 5]])
x[ind]
```

[94]:
```
array([[71, 86],
       [60, 20]])
```

Fancy indexing also works in multiple dimensions. Consider the following array:

[95]:
```
X = np.arange(12).reshape((3, 4))
X
```

[95]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Like with standard indexing, the first index refers to the row, and the second to the column:

[96]:
```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
```

[96]:
```
array([ 2,  5, 11])
```

Notice that the first value in the result is `X[0, 2]`, the second is `X[1, 1]`, and the third is `X[2, 3]`. If we combine a column vector and a row vector within the indices, we get a two-dimensional result:

[97]:
```
X[row[:, np.newaxis], col]
```

[97]:
```
array([[ 2,  1,  3],
       [ 6,  5,  7],
       [10,  9, 11]])
```

Here, each row value is matched with each column vector, exactly as we saw in broadcasting of arithmetic operations. For example:

[98]:
```
row[:, np.newaxis] * col
```

[98]:
```
array([[0, 0, 0],
       [2, 1, 3],
       [4, 2, 6]])
```

It is always important to remember with fancy indexing that the return value reflects the *broadcasted shape of the indices*, rather than the shape of the array being indexed.

5.2 Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen:

```
[99]: print(X)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

We can combine fancy and simple indices:

```
[100]: X[2, [2, 0, 1]]
```

```
[100]: array([10,  8,  9])
```

We can also combine fancy indexing with slicing:

```
[101]: X[1:, [2, 0, 1]]
```

```
[101]: array([[ 6,  4,  5],
              [10,  8,  9]])
```

And we can combine fancy indexing with masking:

```
[102]: mask = np.array([1, 0, 1, 0], dtype=bool)
       X[row[:, np.newaxis], mask]
```

```
[102]: array([[ 0,  2],
              [ 4,  6],
              [ 8, 10]])
```

All of these indexing options combined lead to a very flexible set of operations for accessing and modifying array values.