Numpy Review

October 19, 2016

```
In [1]: from IPython.core.display import HTML
        def css_styling():
            styles = open("styles/custom.css", "r").read()
            return HTML(styles)
        css_styling()
```

```
Out[1]: <IPython.core.display.HTML object>
```

1 NumPy

The most fundamental third-party package for scientific computing in Python is NumPy, which provides multidimensional array data types, along with associated functions and methods to manipulate them. While Python comes with several container types (`list`,`tuple`,`dict`), NumPy's arrays are implemented closer to the hardware, and are therefore more efficient than the built-in types.

1.1 Basics of Numpy arrays

We now turn our attention to the Numpy library, which forms the base layer for the entire 'scipy ecosystem'. Once you have installed numpy, you can import it as

```
In [2]: import numpy
```

though in this book we will use the common shorthand

```
In [3]: import numpy as np
```

As mentioned above, the main object provided by numpy is a powerful array. We'll start by exploring how the numpy array differs from Python lists. We start by creating a simple list and an array with the same contents of the list:

```
In [4]: lst = list(range(1000))
        arr = np.arange(1000)

        # Here's what the array looks like
        arr[:10]
```

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

```
In [5]: type(arr)

Out[5]: numpy.ndarray

In [6]: %timeit [i**2 for i in lst]

1000 loops, best of 3: 463 µs per loop


In [7]: %timeit arr**2

The slowest run took 18.56 times longer than the fastest. This could mean that an i
1000000 loops, best of 3: 1.63 µs per loop
```

Elements of a one-dimensional array are indexed with square brackets, as with lists:

```
In [8]: arr[5:10]

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

In [9]: arr[-1]

Out[9]: 999
```

The first difference to note between lists and arrays is that arrays are *homogeneous*; i.e. all elements of an array must be of the same type. In contrast, lists can contain elements of arbitrary type. For example, we can change the last element in our list above to be a string:

```
In [10]: lst[0] = 'a string inside a list'
         lst[:10]

Out[10]: ['a string inside a list', 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

but the same can not be done with an array, as we get an error message:

```
In [11]: arr[0] = 'a string inside an array'


         ---------------------------------------------------------------------------

         ValueError                                Traceback (most recent call last)

         <ipython-input-11-8ae5b56a752e> in <module>()
    ----> 1 arr[0] = 'a string inside an array'


         ValueError: invalid literal for int() with base 10: 'a string inside an arr
```

The information about the type of an array is contained in its *dtype* attribute:

```
In [12]: arr.dtype

Out[12]: dtype('int64')
```

Once an array has been created, its dtype is fixed and it can only store elements of the same type. For this example where the dtype is integer, if we store a floating point number it will be automatically converted into an integer:

```
In [13]: arr[0] = 1.234
         arr[:10]

Out[13]: array([1, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Above we created an array from an existing list; now let us now see other ways in which we can create arrays, which we'll illustrate next. A common need is to have an array initialized with a constant value, and very often this value is 0 or 1 (suitable as starting value for additive and multiplicative loops respectively); `zeros` creates arrays of all zeros, with any desired dtype:

```
In [14]: np.zeros(5, float)

Out[14]: array([ 0.,  0.,  0.,  0.,  0.])

In [15]: np.zeros(3, int)

Out[15]: array([0, 0, 0])

In [16]: np.zeros(3, complex)

Out[16]: array([ 0.+0.j,  0.+0.j,  0.+0.j])
```

and similarly for `ones`:

```
In [17]: print('5 ones: {0}'.format(np.ones(5)))

5 ones: [ 1.  1.  1.  1.  1.]
```

If we want an array initialized with an arbitrary value, we can create an empty array and then use the fill method to put the value we want into the array:

```
In [18]: a = np.empty(4)
         a

Out[18]: array([  2.31584178e+077,   2.31584178e+077,   2.15554053e-314,
                  2.78136388e-309])

In [19]: a.fill(5.5)
         a

Out[19]: array([ 5.5,  5.5,  5.5,  5.5])
```

```
In [20]: %timeit np.empty(1000)

The slowest run took 11.74 times longer than the fastest. This could mean that an i
1000000 loops, best of 3: 1.12 µs per loop


In [21]: %timeit np.ones(1000)

The slowest run took 10.17 times longer than the fastest. This could mean that an i
100000 loops, best of 3: 3.35 µs per loop
```

We have seen how the `arange` function generates an array for a range of integers. Relatedly, the `linspace` and `logspace` functions to create linearly and logarithmically-spaced grids respectively, with a fixed number of points and including both ends of the specified interval:

```
In [22]: np.linspace(0, 1, num=5)

Out[22]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])

In [23]: np.logspace(1, 4, num=4)

Out[23]: array([    10.,     100.,    1000.,   10000.])
```

Finally, it is often useful to create arrays with random numbers that follow a specific distribution. The `np.random` module contains a number of functions that can be used to this effect, for example this will produce an array of 5 random samples taken from a standard normal distribution (0 mean and variance 1):

```
In [24]: np.random.randn(5)

Out[24]: array([-1.99623594, -0.98083123,  1.31984373, -1.10816431,  1.44163006])
```

whereas this will also give 5 samples, but from a normal distribution with a mean of 10 and a standard deviation of 3:

```
In [25]: norm10 = np.random.normal(loc=10, scale=3, size=10)

         norm10

Out[25]: array([  6.25175593,  11.1878508 ,   7.44616107,   5.9560185 ,
                  9.13689971,  10.83569133,   8.75303915,  13.93881889,
                  6.6465618 ,  14.05655344])
```

1.2 Indexing with other arrays

Above we saw how to index arrays with single numbers and slices, just like Python lists. But arrays allow for a more sophisticated kind of indexing which is very powerful: you can index an array with another array, and in particular with an array of boolean values. This is particluarly useful to extract information from an array that matches a certain condition.

Consider for example that in the array `norm10` we want to replace all values above 9 with the value 0. We can do so by first finding the *mask* that indicates where this condition is true or false:

```
In [26]: mask = norm10 > 9
         mask

Out[26]: array([False,  True, False, False,  True,  True, False,  True, False,  Tru
```

Now that we have this mask, we can use it to either read those values or to reset them to 0:

```
In [27]: norm10[mask]

Out[27]: array([ 11.1878508 ,   9.13689971,  10.83569133,  13.93881889,  14.0565534

In [28]: norm10[mask] = 0

         norm10

Out[28]: array([ 6.25175593,  0.        ,  7.44616107,  5.9560185 ,  0.        ,
                 0.        ,  8.75303915,  0.        ,  6.6465618 ,  0.        ])
```

1.3 Multidimensional Arrays

Numpy can create arrays of aribtrary dimensions, and all the methods illustrated in the previous section work with more than one dimension. For example, a list of lists can be used to initialize a two dimensional array:

```
In [29]: lst2 = [[1, 2], [3, 4]]
         arr2 = np.array([[1, 2], [3, 4]])
         arr2.shape

Out[29]: (2, 2)
```

With two-dimensional arrays we start seeing the power of numpy: while a nested list can be indexed using repeatedly the `[ ]` operator, multidimensional arrays support a much more natural indexing syntax with a single `[ ]` and a set of indices separated by commas:

```
In [30]: lst2[0][1]

Out[30]: 2

In [31]: arr2[0,1]

Out[31]: 2
```

Most of the array creation functions listed above can be used with more than one dimension, for example:

```
In [32]: np.zeros((2,3))

Out[32]: array([[ 0.,  0.,  0.],
                [ 0.,  0.,  0.]])

In [33]: np.random.normal(10, 3, size=(2, 4))
```

```
Out[33]: array([[ 13.48887961,  12.43504832,  10.58748258,   8.21610525],
                [  8.1210449 ,   9.37090385,   6.49745275,  11.3775681 ]])
```

In fact, the shape of an array can be changed at any time, as long as the total number of elements is unchanged. For example, if we want a 2x4 array with numbers increasing from 0, the easiest way to create it is via the numpy array's `reshape` method.

```
In [34]: arr = np.arange(8).reshape(2,4)

         arr

Out[34]: array([[0, 1, 2, 3],
                [4, 5, 6, 7]])
```

With multidimensional arrays, you can also use slices, and you can mix and match slices and single indices in the different dimensions (using the same array as above):

```
In [35]: arr[1, 2:4]

Out[35]: array([6, 7])

In [36]: arr[:, 2]

Out[36]: array([2, 6])
```

If you only provide one index, then you will get the corresponding row.

```
In [37]: arr[1]

Out[37]: array([4, 5, 6, 7])
```

Now that we have seen how to create arrays with more than one dimension, it's a good idea to look at some of the most useful properties and methods that arrays have. The following provide basic information about the size, shape and data in the array:

```
In [38]: print('Data type                :', arr.dtype)
         print('Total number of elements :', arr.size)
         print('Number of dimensions     :', arr.ndim)
         print('Shape (dimensionality)   :', arr.shape)
         print('Memory used (in bytes)   :', arr.nbytes)

Data type                : int64
Total number of elements : 8
Number of dimensions     : 2
Shape (dimensionality)   : (2, 4)
Memory used (in bytes)   : 64
```

Arrays also have many useful methods, some especially useful ones are:

```
In [39]: print('Minimum and maximum             :', arr.min(), arr.max())
         print('Sum and product of all elements :', arr.sum(), arr.prod())
         print('Mean and standard deviation     :', arr.mean(), arr.std())

Minimum and maximum             : 0 7
Sum and product of all elements : 28 0
Mean and standard deviation     : 3.5 2.29128784748
```

For these methods, the above operations area all computed on all the elements of the array. But for a multidimensional array, it's possible to do the computation along a single dimension, by passing the `axis` parameter; for example:

```
In [40]: arr

Out[40]: array([[0, 1, 2, 3],
                [4, 5, 6, 7]])

In [41]: arr.sum(axis=0)

Out[41]: array([ 4,  6,  8, 10])

In [42]: arr.sum(axis=1)

Out[42]: array([ 6, 22])
```

As you can see in this example, the value of the `axis` parameter is the dimension which will be *consumed* once the operation has been carried out. This is why to sum along the rows we use `axis=0`.

This can be easily illustrated with an example that has more dimensions; we create an array with 4 dimensions and shape `(3,4,5,6)` and sum along the axis number 2 (i.e. the *third* axis, since in Python all counts are 0-based). That consumes the dimension whose length was 5, leaving us with a new array that has shape `(3,4,6)`:

```
In [43]: np.zeros((3,4,5,6)).sum(2).shape

Out[43]: (3, 4, 6)
```

Another widely used property of arrays is the `.T` attribute, which allows you to access the transpose of the array:

```
In [44]: arr.T

Out[44]: array([[0, 4],
                [1, 5],
                [2, 6],
                [3, 7]])
```

There is a wide variety of methods and properties of arrays.

```
In [45]: [attr for attr in dir(arr) if not attr.startswith('__')]
```

```
Out[45]: ['T',
          'all',
          'any',
          'argmax',
          'argmin',
          'argpartition',
          'argsort',
          'astype',
          'base',
          'byteswap',
          'choose',
          'clip',
          'compress',
          'conj',
          'conjugate',
          'copy',
          'ctypes',
          'cumprod',
          'cumsum',
          'data',
          'diagonal',
          'dot',
          'dtype',
          'dump',
          'dumps',
          'fill',
          'flags',
          'flat',
          'flatten',
          'getfield',
          'imag',
          'item',
          'itemset',
          'itemsize',
          'max',
          'mean',
          'min',
          'nbytes',
          'ndim',
          'newbyteorder',
          'nonzero',
          'partition',
          'prod',
          'ptp',
          'put',
          'ravel',
          'real',
          'repeat',
```

```
          'reshape',
          'resize',
          'round',
          'searchsorted',
          'setfield',
          'setflags',
          'shape',
          'size',
          'sort',
          'squeeze',
          'std',
          'strides',
          'sum',
          'swapaxes',
          'take',
          'tobytes',
          'tofile',
          'tolist',
          'tostring',
          'trace',
          'transpose',
          'var',
          'view']
```

You can access the documentation for any of them using the `help` function.

```
In [46]: help(arr.strides)

Help on tuple object:

class tuple(object)
 |  tuple() -> empty tuple
 |  tuple(iterable) -> tuple initialized from iterable's items
 |
 |  If the argument is a tuple, the return value is the same object.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
```

```
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __getnewargs__(...)
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __hash__(self, /)
|      Return hash(self).
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.n
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rmul__(self, value, /)
|      Return self*value.
|
|  __sizeof__(...)
|      T.__sizeof__() -- size of T in memory, in bytes
|
|  count(...)
|      T.count(value) -> integer -- return number of occurrences of value
```

```
 |
 |  index(...)
 |      T.index(value, [start, [stop]]) -> integer -- return first index of value.
 |      Raises ValueError if the value is not present.
```

1.4 Array Operations

Arrays support all regular arithmetic operators, and the numpy library also contains a complete collection of basic mathematical functions that operate on arrays. It is important to remember that in general, all operations with arrays are applied *element-wise*, i.e., are applied to all the elements of the array at the same time. Consider for example:

```
In [47]: arr1 = np.arange(4)
         arr2 = np.arange(10, 14)
         arr_sum = arr1 + arr2

         print('{0} + {1} = {2}'.format(arr1, arr2, arr_sum))
```

```
[0 1 2 3] + [10 11 12 13] = [10 12 14 16]
```

Importantly, you must remember that even the multiplication operator is by default applied element-wise, it is *not* the matrix multiplication from linear algebra (as is the case in Matlab, for example):

```
In [48]: print('{0} X {1} = {2}'.format(arr1, arr2, arr1*arr2))
```

```
[0 1 2 3] X [10 11 12 13] = [ 0 11 24 39]
```

While this means that in principle arrays must always match in their dimensionality in order for an operation to be valid, numpy will *broadcast* dimensions when possible. For example, suppose that you want to add the number 1.5 to `arr1`; the following would be a valid way to do it:

```
In [49]: arr1 + 1.5*np.ones(4)
```

```
Out[49]: array([ 1.5,  2.5,  3.5,  4.5])
```

But thanks to numpy's broadcasting rules, the following is equally valid:

```
In [50]: arr1 + 1.5
```

```
Out[50]: array([ 1.5,  2.5,  3.5,  4.5])
```

In this case, numpy looked at both operands and saw that the first (`arr1`) was a one-dimensional array of length 4 and the second was a scalar, considered a zero-dimensional object. The broadcasting rules allow numpy to:

- *create* new dimensions of length 1 (since this doesn't change the size of the array)
- 'stretch' a dimension of length 1 that needs to be matched to a dimension of a different size.

So in the above example, the scalar 1.5 is effectively:

- first 'promoted' to a 1-dimensional array of length 1
- then, this array is 'stretched' to length 4 to match the dimension of `arr1`.

After these two operations are complete, the addition can proceed as now both operands are one-dimensional arrays of length 4.

This broadcasting behavior is in practice enormously powerful, especially because when numpy broadcasts to create new dimensions or to 'stretch' existing ones, it doesn't actually replicate the data. In the example above the operation is carried *as if* the 1.5 was a 1-d array with 1.5 in all of its entries, but no actual array was ever created. This can save lots of memory in cases when the arrays in question are large and can have significant performance implications.

The general rule is: when operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward, creating dimensions of length 1 as needed. Two dimensions are considered compatible when

- they are equal to begin with, or
- one of them is 1; in this case numpy will do the 'stretching' to make them equal.

If these conditions are not met, a `ValueError: frames are not aligned` exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the maximum size along each dimension of the input arrays.

This shows how the broadcasting rules work in several dimensions:

```
In [51]: b = np.array([2, 3, 4, 5])
         bcast_sum = arr + b

         print('{0}\n\n+ {1}\n{2}\n{3}'.format(arr, b, '-'*12, bcast_sum))
```

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

+ [2 3 4 5]
------------
[[ 2  4  6  8]
 [ 6  8 10 12]]
```

Now, how could you use broadcasting to say add `[4, 6]` along the rows to `arr` above? Simply performing the direct addition will produce the error we previously mentioned:

```
In [52]: c = np.array([4, 6])
         arr + c
```

```
---------------------------------------------------------------------------
```

```
ValueError                                Traceback (most recent call last)

<ipython-input-52-62aa20ac1980> in <module>()
  1 c = np.array([4, 6])
----> 2 arr + c


ValueError: operands could not be broadcast together with shapes (2,4) (2,)
```

According to the rules above, the array `c` would need to have a *trailing* dimension of 1 for the broadcasting to work. It turns out that numpy allows you to 'inject' new dimensions anywhere into an array on the fly, by indexing it with the special object `np.newaxis`:

```
In [53]: c

Out[53]: array([4, 6])

In [54]: cplus = c[:, np.newaxis, np.newaxis]
         np.squeeze(cplus)

Out[54]: array([4, 6])
```

This is exactly what we need, and indeed it works:

```
In [55]: arr + cplus

Out[55]: array([[[ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]],

                [[ 6,  7,  8,  9],
                 [10, 11, 12, 13]]])
```

For the full broadcasting rules, please see the official Numpy docs, which describe them in detail and with more complex examples.

1.4.1 Exercises

Generate the following structure as a numpy array, without typing the values by hand. Then, create another array containing just the 2nd and 4th rows.

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

```
In [56]: # Write your answer here
```

Divide each column of the array:

```
a = np.arange(25).reshape(5, 5)
```

elementwise with the array `b = np.array([1., 5, 10, 15, 20])`.
Hint: `np.newaxis`

```
In [57]: # Write your answer here
```

Generate a 10 x 3 array of random numbers (in range [0,1]). For each row, pick the number closest to 0.5.
Hints:

- NumPy functions/methods that may be useful here include `abs`, `choose` and `argsort`

```
In [58]: # Write your answer here
```

1.5 Linear Algebra

Numpy ships with a basic linear algebra library, and all arrays have a `dot` method whose behavior is that of the scalar dot product when its arguments are vectors (one-dimensional arrays) and the traditional matrix multiplication when one or both of its arguments are two-dimensional arrays:

```
In [59]: v1 = np.array([2, 3, 4])
         v2 = np.array([1, 0, 1])
         dprod = v1.dot(v2)

         print(v1, '.', v2, '=', dprod)
```

```
[2 3 4] . [1 0 1] = 6
```

There is an equivalent `dot` function:

```
In [60]: np.dot(v1, v2)
```

```
Out[60]: 6
```

Here is a regular matrix-vector multiplication, note that the array `v1` should be viewed as a *column* vector in traditional linear algebra notation; numpy makes no distinction between row and column vectors and simply verifies that the dimensions match the required rules of matrix multiplication, in this case we have a 2×3 matrix multiplied by a 3-vector, which produces a 2-vector:

```
In [61]: A = np.arange(6).reshape(2, 3)
         A.dot(v1)
```

```
Out[61]: array([11, 38])
```

For matrix-matrix multiplication, the same dimension-matching rules must be satisfied, e.g. consider the difference between $A \times A^T$:

```
In [62]: A.dot(A.T)

Out[62]: array([[ 5, 14],
                [14, 50]])
```

and $A^T \times A$:

```
In [63]: A.T.dot(A)

Out[63]: array([[ 9, 12, 15],
                [12, 17, 22],
                [15, 22, 29]])
```

Furthermore, the `numpy.linalg` module includes additional functionality such as determinants, matrix norms, Cholesky, eigenvalue and singular value decompositions, etc. For even more linear algebra tools, `scipy.linalg` contains the majority of the tools in the classic LAPACK libraries as well as functions to operate on sparse matrices. We refer the reader to the Numpy and Scipy documentations for additional details on these.

1.6 Reading and writing arrays to disk

Numpy lets you read and write arrays into files in a number of ways. In order to use these tools well, it is critical to understand the difference between a *text* and a *binary* file containing numerical data. In a text file, the number π could be written as "3.141592653589793", for example: a string of digits that a human can read, with in this case 15 decimal digits. In contrast, that same number written to a binary file would be encoded as 8 characters (bytes) that are not readable by a human but which contain the exact same data that the variable `pi` had in the computer's memory.

The tradeoffs between the two modes are thus:

- **Text mode**: occupies more space, precision can be lost (if not all digits are written to disk), but is readable and editable by hand with a text editor. Can *only* be used for one- and two-dimensional arrays.
- **Binary mode**: compact and exact representation of the data in memory, can't be read or edited by hand. Arrays of any size and dimensionality can be saved and read without loss of information.

First, let's see how to read and write arrays in text mode. The `np.savetxt` function saves an array to a text file, with options to control the precision, separators and even adding a header:

```
In [64]: arr = np.arange(10).reshape(2, 5)
         np.savetxt('test.out', arr, fmt='%.2e', header="My dataset")

In [65]: !cat test.out

# My dataset

0.00e+00 1.00e+00 2.00e+00 3.00e+00 4.00e+00

5.00e+00 6.00e+00 7.00e+00 8.00e+00 9.00e+00
```

And this same type of file can then be read with the matching `np.loadtxt` function:

```
In [66]: arr2 = np.loadtxt('test.out')
         arr2

Out[66]: array([[ 0.,  1.,  2.,  3.,  4.],
                [ 5.,  6.,  7.,  8.,  9.]])
```

For binary data, Numpy provides the `np.save` and `np.savez` routines. The first saves a single array to a file with `.npy` extension, while the latter can be used to save a *group* of arrays into a single file with `.npz` extension. The files created with these routines can then be read with the `np.load` function.

Let us first see how to use the simpler `np.save` function to save a single array:

```
In [67]: np.save('test.npy', arr2)

         # Now we read this back
         arr2n = np.load('test.npy')

         # Let's see if any element is non-zero in the difference.
         # A value of True would be a problem.
         np.any(arr2 - arr2n)

Out[67]: False
```

Now let us see how the `np.savez` function works. You give it a filename and either a sequence of arrays or a set of keywords. In the first mode, the function will auotmatically name the saved arrays in the archive as `arr_0`, `arr_1`, etc:

```
In [68]: np.savez('test.npz', arr, arr2)
         arrays = np.load('test.npz')
         arrays.files

Out[68]: ['arr_0', 'arr_1']

In [69]: arrays

Out[69]: <numpy.lib.npyio.NpzFile at 0x105ce14a8>
```

Alternatively, we can explicitly name the arrays we save using keyword arguments for `savez`:

```
In [70]: np.savez('test.npz', array1=arr, array2=arr2)
         arrays = np.load('test.npz')
         arrays.files

Out[70]: ['array1', 'array2']
```

The object returned by `np.load` from an `.npz` file works like a dictionary, though you can also access its constituent files by attribute using its special `.f` field; this is best illustrated with an example with the `arrays` object from above:

```
In [71]: # First row of array
         arrays['array1'][0]

Out[71]: array([0, 1, 2, 3, 4])
```

Equivalently:

```
In [72]: arrays.f.array1[0]

Out[72]: array([0, 1, 2, 3, 4])
```

This `.npz` format is a very convenient way to package compactly and without loss of information, into a single file, a group of related arrays that pertain to a specific problem. At some point, however, the complexity of your dataset may be such that the optimal approach is to use one of the standard formats in scientific data processing that have been designed to handle complex datasets, such as NetCDF or HDF5.

1.7 References

Scientific Python Lecture Notes