NumPy, SciPy, Matplotlib, and Pillow: An Insufficient Primer

Ronald T Kneusel
rkneuselbooks@gmail.com

version 1.0

November 6, 2022

Contents

This brief document introduces the parts of NumPy, SciPy, Matplotlib, and Pillow that are most frequently used in my books. Use it as a stepping stone. See the Resources section at the end for places to learn more. The reference websites are

NumPy	`www.numpy.org`
SciPy	`www.scipy.org`
Matplotlib	`www.matplotlib.org`
Pillow	`python-pillow.org`

You'll find installation instructions there for your specific operating system. On Ubuntu, you can usually get by with

```
> pip3 install numpy
> pip3 install scipy
```

```
> pip3 install matplotlib
> pip3 install pillow
```

1 NumPy

NumPy adds array processing to Python. Frankly, without NumPy, Python would be unattractive as a scientific programming language. With NumPy, the sky's the limit.

NumPy is universally imported in the following way:

```
import numpy as np
```

The following presents commands and output from Python. Use it as a guide, but experiment on your own as well.

1.1 Defining Arrays

Let's begin by defining vectors and matrices by hand,

```
>>> import numpy as np
>>> a = np.array([1,2,3,4])
>>> a
array([1, 2, 3, 4])
>>> a.size
4
>>> a.shape
(4,)
>>> a.dtype
dtype('int64')
>>> b = np.array([1,2,3,4], dtype="uint8")
>>> b.dtype
dtype('uint8')
>>> c = np.array([1,2,3,4], dtype="float64")
>>> c.dtype
dtype('float64')
```

Use `np.array` to define arrays from lists. NumPy arrays are typed and follow the data types used by C; therefore, `uint8` is an unsigned 8-bit integer (a byte), and `int64` is a signed 64-bit integer. Use `float32` for a C **`float`** and `float64` for a C `double` (a Python **`float`**).

Array attributes include `size` and `shape`. Alternatively, **`len`** will return the number of elements in the outermost axis of the array (the length, if a vector).

The above creates vectors. Here's how to create matrices and 3D arrays,

```
>>> d = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> d.shape
(3, 3)
>>> d.size
34
>>> d
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> d = np.array([[[1,11,111],[2,22,222]],[[3,33,333],[4,44,444]]])
```

```
>>> d.shape
(2, 2, 3)
>>> d
array([[[ 1, 11, 111],
        [ 2, 22, 222]],

       [[ 3, 33, 333],
        [ 4, 44, 444]]])
```

Create larger arrays with `zeros` and `ones`

```
>>> x = np.zeros((2,3,4))
>>> x.shape
(2, 3, 4)
>>> x.dtype
dtype('float64')
>>> b = np.zeros((10,10),dtype="uint32")
>>> b.shape
(10, 10)
>>> b.dtype
dtype('uint32')
>>> y = np.ones((3,3))
>>> y
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
>>> y = 10*np.ones((3,3))
>>> y
array([[10., 10., 10.],
       [10., 10., 10.],
       [10., 10., 10.]])
>>> y.dtype
dtype('float64')
>>> y.astype("uint8")
array([[10, 10, 10],
       [10, 10, 10],
       [10, 10, 10]], dtype=uint8)
```

NumPy arrays are assigned by reference unless explicit action is taken to copy the array. Consider,

```
>>> a = np.range(10)
>>> b = a
>>> c = a.copy()
>>> d = a[:]
```

Here, `b` points to `a`, meaning any changes to `a` will be reflected in `b` as it's merely another name for the same memory. However, `c` and `d` are copies of `a`, the latter copying by indexing all elements.

1.2 Indexing Arrays

Arrays are indexed via square brackets. For example,

```
>>> b = np.zeros((3,4),dtype='uint8')
>>> b
array([[0,0,0,0],
```

```
       [0,0,0,0],
       [0,0,0,0], dtype=uint8)
>>> b[0,1] = 1
>>> b[1,0] = 2
>>> b
array([[0,1,0,0],
       [2,0,0,0],
       [0,0,0,0], dtype=uint8)
>>> b[1,0]
2
>>> b[1]
array([2, 0, 0, 0], dtype=uint8)
>>> b[1][0]
2
```

The first two assignments to `b` update two elements, the first at row 0, column 1, and the second at row 1, column 0. Referring to an element by both indices returns it. Referring to only the first index returns the entire row, i.e., asking for row 1, all columns.

A few more indexing examples introducing slicing,

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[1:4]
array([1, 2, 3])
>>> a[3:7]
array([3, 4, 5, 6])
>>> a[0:8:2]
array([0, 2, 4, 6])
>>> a[3:7:2]
array([3, 5])
>>> a[:6]
array([0, 1, 2, 3, 4, 5])
>>> a[6:]
array([6, 7, 8, 9])
```

The `arange` function is the NumPy equivalent to Python's **`range`** function. Slices use colons to specify the first index and *one more than* the final index. In other words, `[1:4]` selects all elements i such that $1 \leq i < 4$.

If two colons are used, the slice is starting index, ending index plus one, and increment so that `[0:8:2]` returns every element from 0 through but not including 8, incrementing by 2. Slices with no initial index use 0. Slices with no final index run to the end of the axis.

Shortcuts and negative indices are allowed,

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

A negative index counts from the end of the respective axis; therefore, `a[-1]` will always return the last element of a vector. A negative index as an increment counts down so that `[::-1]` has the effect of reversing a vector or axis where `::-1` is used.

Slicing works across multiple axes,

```
>>> b = np.arange(20).reshape((4,5))
>>> b
array([[ 0, 1, 2, 3, 4],
       [ 5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> b[1:3,:]
array([[ 5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
>>> b[2:,2:]
array([[12, 13, 14],
       [17, 18, 19]])
```

Notice the call to the `reshape` method to transform a vector into a 4x5 array.

Let's review a few more indexing examples. First, define some arrays,

```
>>> c = np.arange(27).reshape((3,3,3))
>>> c
array([[[ 0, 1, 2],
        [ 3, 4, 5],
        [ 6, 7, 8]],
       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],
        [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
>>> a = np.ones((3,3))
>>> a
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Then, index using colon to select subarrays,

```
>>> c[1,:,:] = a
>>> c
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],
       [[ 1,  1,  1],
        [ 1,  1,  1],
        [ 1,  1,  1]],
       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

Notice that the 3x3 subarray at `c[1]` has been updated with `a`. It's convenient to think of a 3D array as a stack of 2D arrays, here a stack of three 3x3 arrays.

The same indexing can be accomplished with the ellipsis, which stands for "as many colons as needed,"

```
>>> c[0,...] = a
>>> c
array([[[ 1,  1,  1],
        [ 1,  1,  1],
        [ 1,  1,  1]],
```

```
       [[ 1,  1,  1],
        [ 1,  1,  1],
        [ 1,  1,  1]],
       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

1.3 Operators and Broadcasting

NumPy supports the expected Python operators along with a notion called "broadcasting" where NumPy will, wisely, figure out what you want to do and just do it without explicit loops. Some examples,

```
>>> a = np.arange(5)
>>> a
array([ 0, 1, 2, 3, 4])
>>> c = np.arange(5)[::-1]
>>> c
array([ 4, 3, 2, 1, 0])
>>> a*3.14
array([ 0., 3.14, 6.28, 9.42, 12.56])
>>> a*a
array([ 0, 1, 4, 9, 16])
>>> a*c
array([0, 3, 4, 3, 0])
>>> a//(c+1)
array([0, 0, 0, 1, 4])
```

NumPy broadcasts across the arrays as needed. Here's a more complex broadcast example,

```
>>> a
array([0, 1, 2, 3, 4])
>>> b = np.arange(30).reshape((6,5))
>>> b
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29]])
>>> a*b
array([[  0,   1,   4,   9,  16],
       [  0,   6,  14,  24,  36],
       [  0,  11,  24,  39,  56],
       [  0,  16,  34,  54,  76],
       [  0,  21,  44,  69,  96],
       [  0,  26,  54,  84, 116]])
```

The vector, `a`, has five elements. The vector, `b` is, via `reshape`, turned into a 6x5 matrix. NumPy sees the expression `a*b` and understands that the five-element vector matches the five columns of `b`, so it broadcasts `a` across each row of `b`, no loop required.

NumPy supports matrix math,

```
>>> a = np.arange(9).reshape((3,3))
```

```
>>> b = np.arange(9).reshape((3,3))
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.dot(a,b)
array([[ 15, 18, 21],
       [ 42, 54, 66],
       [ 69, 90, 111]])
>>> a*b
array([[ 0, 1, 4],
       [ 9, 16, 25],
       [36, 49, 64]])
```

The call to `dot` uses matrix multiplication to multiply the two 3x3 matrices, `a` and `b`. Notice that this is not standard elementwise multiplication. The matrix multiply operator (`@`) works here as well,

```
>>> a @ b
array([[ 15,  18,  21],
       [ 42,  54,  66],
       [ 69,  90, 111]])
```

1.4 Array Input/Output

For this section, we need a data file, call it *abc.txt* and give it the following three lines with spaces between the values,

```
1 2 3
4 5 6
7 8 9
```

Now, make a second version, *abc.csv*,

```
1,2,3
4,5,6
7,8,9
```

Here's how to load such files in NumPy,

```
>>> a = np.loadtxt("abc.txt")
>>> a
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
>>> a = np.loadtxt("abc.csv", delimiter=",")
>>> a
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
>>> np.save("abc.npy", a)
>>> b = np.load("abc.npy")
>>> b
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

```
>>> np.savetxt("ABC.txt", b)
>>> np.savetxt("ABC.csv", b, delimiter=",")
```

The NumPy function, `loadtxt`, parses text file input. By default, it assumes spaces or tabs between values on a line. To load *abc.csv*, tell NumPy that a comma is the delimiter. You'll also find the `skiprows` keyword helpful to skip any header lines before the data.

To write an array to disk in NumPy format, use the `save` function with the filename as the first argument. The .npy extension is standard. The `load` function reads the array from disk. Note, `save` and `load` work with single arrays. To store multiple arrays in a single file, use `savez`,

```
>>> a
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
>>> b
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
>>> np.savez("arrays.npz", a=a, b=b)
>>> q = np.load("arrays.npz")
>>> list(q.keys())
['a', 'b']
>>> q['a']
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
>>> q['b']
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
```

NumPy's `load` knows the difference between .npy and .npz files. In the latter case, it returns a dictionary where the keys are the names supplied in the `savez` call.

1.5 Random Numbers

NumPy can generate all manner of pseudorandom numbers via its `np.random` module. The most common is uniformly distributed numbers in $[0, 1)$,

```
>>> np.random.random(5)
array([0.89058646, 0.58518431, 0.04111247, 0.91078347, 0.43307253])
>>> np.random.random((3,4))
array([[0.37933472, 0.90327196, 0.69033026, 0.22802077],
       [0.35824227, 0.61660655, 0.79275361, 0.90113857],
       [0.91599903, 0.49936666, 0.36592385, 0.99835758]])
>>> np.random.seed(8675309)
>>> np.random.random(5)
array([0.81245912, 0.75104509, 0.35736652, 0.20229478, 0.40900113])
>>> np.random.seed(8675309)
>>> np.random.random(5)
array([0.81245912, 0.75104509, 0.35736652, 0.20229478, 0.40900113])
```

Notice the call to `seed`? It sets the global pseudorandom number seed. This is dangerous to do in actual code, especially when using things like Scikit-Learn that also use NumPy's pseudorandom generators. All the same, for many examples in my books, I live dangerously and use these functions. Caveat emptor.

The recommended way to use random numbers in NumPy is to use one of the supported pseudorandom generators directly,

```
>>> seed = 73939133
>>> pcg64 = np.random.Generator(np.random.PCG64(seed))
>>> mt19937 = np.random.Generator(np.random.MT19937(seed))
>>> mt19937.random(5)
array([0.70140818, 0.65138414, 0.13270157, 0.36737531, 0.17020404])
>>> pcg64.random(4)
array([0.84873609, 0.56954764, 0.54313798, 0.27999877])
```

When initializing the generator, a `seed` value may be supplied.

1.6 Useful Functions and Methods

Here are a collection of useful NumPy functions (in the sense that I use them often in my books):

```
>>> t = np.random.normal(size=10000)
>>> h,x = np.histogram(t, bins=30)
>>> h
array([   2,    3,   10,   26,   51,   86,  147,  227,  330,  470,  631,
        795,  869,  995, 1022, 1017,  892,  721,  557,  430,  284,  202,
        119,   51,   31,   16,   12,    2,    1,    1])
>>> x = 0.5*(x[1:]+x[:-1])
>>> x
array([-3.49447494, -3.24112542, -2.98777589, -2.73442637, -2.48107685,
       -2.22772732, -1.9743778 , -1.72102828, -1.46767875, -1.21432923,
       -0.96097971, -0.70763018, -0.45428066, -0.20093114,  0.05241838,
        0.30576791,  0.55911743,  0.81246695,  1.06581648,  1.319166  ,
        1.57251552,  1.82586505,  2.07921457,  2.33256409,  2.58591361,
        2.83926314,  3.09261266,  3.34596218,  3.59931171,  3.85266123])
```

`Histogram`, as its name suggests, creates histograms returning first the bin counts (`h`) followed by the bin edge positions (`x`). For plotting, it's often convenient to replace `x` by the midpoint of each bin.

If the vector is integer-valued, `bincount` is your friend,

```
>>> t = np.random.randint(0,100,size=10000)
>>> np.bincount(t, minlength=100)
array([111,  86, 117, 107,  80,  96, 103,  89, 101,  83, 115, 114, 104,
        94,  79,  96, 102,  98, 111, 115,  92,  93,  93, 107, 106,  88,
        80, 106,  92,  91,  86, 105, 101, 109, 120, 105, 103,  91, 106,
        98, 114, 101,  96, 106,  95, 102, 101,  95, 106,  90, 114,  94,
        95,  96, 103,  94, 112,  99,  95, 103, 112,  93, 105, 101,  98,
        95,  82, 127,  90,  96, 105,  97, 107,  90, 115, 114, 106, 117,
        92, 118, 111,  96, 110,  82,  93, 114,  95,  86,  99,  85, 102,
       107,  83,  92,  99, 102,  97,  86, 109, 108])
```

`Bincount` returns a count of the unique values in the vector, so the index is the value, i.e., 2 appeared 117 times in `t`. Use `minlength` to return 0 for values that don't appear but might have,

```
>>> t = np.array([0,4,3,2,5,2,7,7,4,2,3,1,0,2])
>>> np.bincount(t)
array([2, 1, 4, 2, 2, 1, 0, 2])
>>> np.bincount(t, minlength=10)
array([2, 1, 4, 2, 2, 1, 0, 2, 0, 0])
```

In the first case, the maximum value in `t` is 7, so counts for values from 0 through 7 are returned. By adding `minlength`, zeros are returned for 8 and 9, which don't appear in `t` but might have.

To flatten an array, use `ravel`, which is a contronym, but in this case means "unravel,"

```
>>> x = np.arange(27).reshape((3,3,3))
>>> x
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
>>> x.ravel()
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26])
```

To remove dimensions of 1 from an array, use `squeeze`,

```
>>> w = np.arange(4).reshape((1,4))
>>> w
array([[0, 1, 2, 3]])
>>> w.shape
(1, 4)
>>> w.squeeze()
array([0, 1, 2, 3])
>>> w.squeeze().shape
(4,)
```

It is often necessary, especially when working with deep learning toolkits like Keras, to add new dimensions of size 1. For that, use `np.newaxis`,

```
>>> w = np.arange(12).reshape((4,3))
>>> w
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> w.shape
(4, 3)
>>> x = w[:,np.newaxis,:]
>>> x
array([[[ 0,  1,  2]],

       [[ 3,  4,  5]],
```

```
       [[ 6,  7,  8]],

       [[ 9, 10, 11]]])
>>> x.shape
(4, 1, 3)
>>> w[3,2]
11
>>> x[3,0,2]
11
```

Note, `np.newaxis` is exactly `None`, so you might see that used as well.

NumPy will give you basic stats on an array,

```
>>> t = np.random.normal(5,2,size=100)
>>> t.mean(), t.std()
(5.2998969178412025, 1.9190509592773324)
>>> t.std(ddof=1)
1.9287187834831852
>>> np.median(t)
5.3223284095557135
```

`Mean` returns the mean and `std` returns the standard deviation (σ), the square root of the variance. By default, NumPy returns the *population* standard deviation, also known as the *biased* estimate,

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(x_i - \mu)^2}$$

Here, μ is the population mean.

However, most stats packages return the *unbiased* estimate, i.e., the *sample* standard deviation,

$$s = \sqrt{\frac{1}{N-1}\sum_{i=0}^{N-1}(x_i - \bar{x})^2}$$

This is what `std(ddof=1)` returns.

Use `median` to get the median of the array; as with `mean`, the shape doesn't matter. Recall the median is the middle value, the 50-th percentile. Half the values are below the median, and half are above. Use the median for distributions that are not symmetric, as the mean is often misleading.

To get other percentiles, use `percentile` or `quantile`,

```
>>> t = np.random.normal(size=10000)
>>> np.median(t)
0.0027472720968729127
>>> np.percentile(t, 25), np.percentile(t, 75)
(-0.6702662468312279, 0.6838417099373242)
>>> np.quantile(t, 0.25), np.quantile(t, 0.75)
(-0.6702662468312279, 0.6838417099373242)
```

Finally, a common task in machine learning is to standardize a dataset. Datasets are typically stored as a matrix where the rows are the samples (think classical machine learning) and the columns are the features. To standardize, the mean of each feature must be subtracted before dividing by the standard deviation of the feature values.

To get per feature means and standard deviations, use the `axis` keyword to project along the row (axis 0),

```
>>> d = np.random.randint(1,7,size=(5,10))
>>> d
array([[1, 1, 3, 2, 1, 4, 6, 1, 2, 1],
       [4, 2, 4, 1, 4, 6, 2, 1, 5, 6],
       [1, 6, 5, 5, 2, 6, 4, 2, 2, 5],
       [1, 2, 3, 6, 4, 5, 2, 5, 4, 3],
       [5, 5, 6, 1, 4, 3, 1, 2, 3, 5]])
>>> d.mean(axis=0)
array([2.4, 3.2, 4.2, 3. , 3. , 4.8, 3. , 2.2, 3.2, 4. ])
>>> d.std(ddof=1, axis=0)
array([1.94935887, 2.16794834, 1.30384048, 2.34520788, 1.41421356,
       1.30384048, 2.        , 1.64316767, 1.30384048, 2.        ])
```

Standardize using these means and standard deviations, along with NumPy's broadcasting rules,

```
>>> s = (d - d.mean(axis=0)) / d.std(ddof=1, axis=0)
>>> s.mean(axis=0)
array([ 4.44089210e-17, -1.11022302e-16, -1.33226763e-16,  4.44089210e-17,
        0.00000000e+00,  1.33226763e-16,  0.00000000e+00, -1.24900090e-16,
       -1.33226763e-16,  0.00000000e+00])
>>> s.std(ddof=1, axis=0)
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Standardization makes the feature means zero and the feature standard deviations one.

2 SciPy

SciPy is an extensive package. My typical use is restricted to hypothesis testing, though other functions show up from time to time. For hypothesis testing, I use `ttest_ind` for the independent t-test and `mannwhitneyu` for the nonparametric Mann-Whitney U test:

```
>>> a = np.random.normal(1.7,4,size=220)
>>> b = np.random.normal(1,2,size=220)
>>> ttest_ind(a,b)
Ttest_indResult(statistic=2.570856534819965, pvalue=0.010473935577379087)
>>> mannwhitneyu(a,b)
MannwhitneyuResult(statistic=27945.0, pvalue=0.004990491890815961)
```

Cohen's d is also useful for measuring effect size. It's easily defined,

```
def Cohen_d(a,b):
    s1 = np.std(a, ddof=1)**2
    s2 = np.std(b, ddof=1)**2
    return (a.mean() - b.mean()) / np.sqrt(0.5*(s1+s2))
```

which, for `a` and `b` above, returns,

```
>>> Cohen_d(a,b)
0.24512155282683945
```

indicating a small effect size.

3 Matplotlib

`Matplotlib` is Python's standard graphing library. A typical plot runs like this,

```
>>> import matplotlib.pylab as plt
>>> x = np.linspace(-5,5,30)
>>> y = x**3 + 1
>>> plt.plot(x,y, marker='o', color='k')
[<matplotlib.lines.Line2D object at 0x7f1a78357a60>]
>>> plt.xlabel("x")
Text(0.5, 0, 'x')
>>> plt.ylabel("y")
Text(0, 0.5, 'y')
>>> plt.show()
```

First, make the `plot`, then `show` the plot. The displayed plot is interactive.

In most cases, plotting is used to generate figures for the books and run from a script. First, create this file,

```
import numpy as np
import matplotlib.pylab as plt
x = np.linspace(-5,5,30)
y0 = 0.5*x**3 + 1
y1 = -x**2 + 4
plt.plot(x,y0, marker='o', color='k', label='0.5*x**3+1')
plt.plot(x,y1, marker='s', fillstyle='none', color='#2391ae', label='-x**2+4
    ')
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.legend(loc='best')
plt.tight_layout(pad=0, w_pad=0, h_pad=0)
plt.savefig('plot.png', dpi=300)
plt.close()
```

then execute it to produce *plot.png* (Figure 1). `Savefig`'s output format is specified by the file extension with `dpi` controlling the output size.

The code defines two functions and then uses `plot` to show them. There are many possible plot keywords, but only a few are shown. For colors, use letters (r,g,b,c,m,k), or specify colors using HTML RGB syntax. To remove the lines, use `linestyle='none'`. Finally, notice the use of `legend` to place the `labels` on the plot automatically in the best place (whatever that means).

An alternative to `plot` is `scatter`, which doesn't use lines,

```
>>> x,y = np.random.random(100), np.random.random(100)
>>> plt.scatter(x,y, marker='+')
<matplotlib.collections.PathCollection object at 0x7fc87c2fda00>
>>> plt.show()
```

Use `bar` to make bar plots,

```
>>> t = np.random.normal(3,2,size=1000)
>>> h,x = np.histogram(t, bins=30)
>>> x = 0.5*(x[1:]+x[:-1])
>>> plt.bar(x,h, width=0.8*(x[1]-x[0]))
<BarContainer object of 30 artists>
>>> plt.show()
```

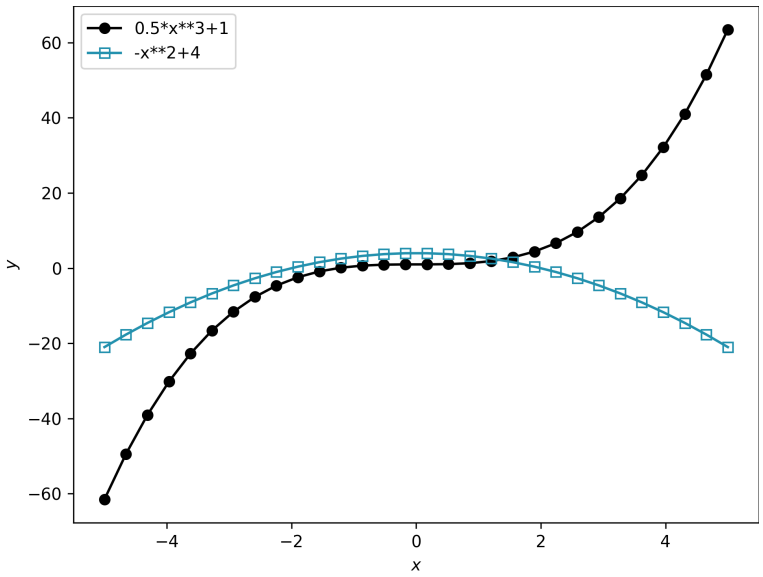

Figure 1: A sample plot

Here, `width` controls the width of the bars, which is set to 80 percent of the distance between successive bin center positions.

4 Pillow

Pillow handles images which are often turned into NumPy arrays and vice versa. The examples in this section use Scikit-Learn,

```
> pip3 install scikit-learn
```

Scikit-Learn supplies traditional machine learning classes along with classic datasets.

Here's how to load an image, convert to and from NumPy arrays, and write images to disk,

```
>>> from PIL import Image
>>> from sklearn.datasets import load_sample_images
>>> china = load_sample_images().images[0]
>>> flower = load_sample_images().images[1]
>>> china.shape, china.dtype
((427, 640, 3), dtype('uint8'))
>>> flower.shape, flower.dtype
((427, 640, 3), dtype('uint8'))
>>> imChina = Image.fromarray(china)
>>> imFlower = Image.fromarray(flower)
>>> imChina.show()
>>> imFlower.show()
>>> imChina.save("china.png")
>>> imFlower.save("flower.jpg")
>>> im = Image.open("china.png")
>>> im.show()
```

```
>>> img = np.array(im)
>>> img.shape, img.dtype
((427, 640, 3), dtype('uint8'))
>>> gray = im.convert("L")
>>> gray.show()
>>> gray.size
(640, 427)
>>> g = np.array(gray)
>>> g.shape, g.dtype
((427, 640), dtype('uint8'))
```

Convert a Pillow `Image` to a NumPy array with `array`. Go the other way by passing the array to `Image.fromarray`. The `dtype` should be `uint8` (unsigned bytes). The three color planes are red, green, and blue for RGB images.

Use `convert` on an `Image` to change it to/from RGB, RGBA, and grayscale (L, for luminance). The alpha channel is A. It's a good idea when loading an image and RGB is wanted to pass it through `convert` to make sure any alpha channel is removed,

```
im = np.array(Image.open("some_image.png").convert("RGB"))
```

When writing an image to disk, specify the file type by using one of the standard extensions.

Notice, when an image is in an `Image` object, `size` returns the number of columns followed by the number of rows. However, when the same image is a NumPy array, the two are flipped so that `shape` returns rows then columns, followed by 3 if an RGB image.

Many times, after manipulating an image as a NumPy array, the data values are no longer in the range $[0, 255]$ and the image is no longer of data type `uint8`. To map back to an image, first scale the image $[0, 1]$, then multiply by 255, and finally convert to `uint8` before passing the array to `Image.fromarray`,

```
>>> im = np.array(Image.open("china.png").convert("L"))
>>> Image.fromarray(im).save("china1.png")
>>> img = im**2.0
>>> img.min(), img.max()
(0.0, 65025.0)
>>> img = img / img.max()
>>> img.min(), img.max()
(0.0, 1.0)
>>> Image.fromarray((255*img).astype("uint8")).save("china2.png")
>>> im3 = im**3.0
>>> im3 = im3 / im3.max()
>>> Image.fromarray((255*im3).astype("uint8")).save("china3.png")
```

This sequence of instructions creates three grayscale versions of the China image: the image as it is, the image squared, and the image cubed. Squaring and cubing bring out detail in the bright background structures.

5 Resources

There are many ways to help you improve your NumPy, SciPy, and Matplotlib skills. Here are just a few:

SciPy Lecture Notes (many authors)

This free book covers NumPy, SciPy, and Matplotlib. Get it here,
`https://scipy-lectures.org/_downloads/ScipyLectures-simple.pdf`

NumPy Tutorial (with colab support)

This tutorial includes colab support so you can try things interactively,
`https://cs231n.github.io/python-numpy-tutorial/`

Numerical Python by Robert Johansson

This popular book covers everything as well.

Pillow Handbook (online)

The official Pillow documentation contains a handbook with a tutorial,
`https://pillow.readthedocs.io/en/stable/handbook/`