Numerical Libraries

Sandro Cumani

sandro.cumani@polito.it

Politecnico di Torino

Numerical libraries

We will make heavy use of numerical libraries

The main ones are `numpy` and `scipy` — for more information:

https://numpy.org/doc/stable/user/quickstart.html
https://numpy.org/doc/stable/index.html

- Multi-dimensional arrays
- Linear algebra
- Numerical optimization
- Many ML algorithms (in particular with `sklearn` – **we won't use pre-built ML tools in most cases**)

Data visualization will rely on `matplotlib`

Numpy

Numpy main objects are multidimensional arrays (`ndarray` class)

We will mainly use one-dimensional (vector) and two-dimensional (matrices) arrays (not to be confused with the `(matrix)` class)

The dimensions are also called axes

- One-dimensional arrays have one axis, two-dimensional arrays have two axes, and so on

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

Numpy

Arrays have attributes that describe the array itself

- `ndarray.size`: Total number of elements
- `ndarray.shape`: Tuple with the number of elements for each axis — A $m \times n$ matrix will have a shape `(m, n)`
- `ndarray.ndim`: Number of axes
- `ndarray.dtype`: The data type (for example, `numpy.int32`, `numpy.float32`, `numpy.float64`

Numpy

```
>>> x = numpy.array([1,2,3])
>>> x.size
3
>>> x.shape
(3,)
>>> y = numpy.array([[1,2,3], [4,5,6]])
>>> y.shape
(2, 3)
>>> y.size
6
>>> y.ndim
2
```

Numpy

Arrays can be created in many ways

- From Python lists or tuples. To create multi-dimensional arrays, use nested lists (see previous examples). We can specify the data type passing the argument `dtype`:

```
>>> numpy.array([1,2,3], dtype=numpy.float64)
array([ 1.,  2.,  3.])
```

- As a copy of another array

```
>>> y = numpy.array(x)
```

Numpy

Arrays can be created in many ways

- Using functions that create predefined arrays: `zero`, `ones`, `arange`, `eye`, ...

```
>>> numpy.zeros((2, 3), dtype=numpy.float32)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]], dtype=float32)

>>> numpy.ones(5)
array([ 1.,  1.,  1.,  1.,  1.])

>>> numpy.arange(4)
array([0, 1, 2, 3])

>>> numpy.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Numpy

The `arange` function is similar to the `range` function, and allows specifying a step size

```
>>> numpy.arange(0, 6, 2)
array([0, 2, 4])
```

If we want to create an array of evenly spaced values in a range, we can use `linspace`

```
>>> numpy.linspace(0, 5, 4)
array([ 0.        ,  1.66666667,  3.33333333,  5.
           ])
```

Numpy

Arithmetic operators on numpy array operate *element-wise*

They require arrays with matching shapes

```
>>> x = numpy.array([[1,2,3], [4,5,6]])
>>> y = numpy.array([[2,2,2], [3,3,3]])
>>> x + y
array([[3, 4, 5],
       [7, 8, 9]])
>>> x * y
array([[ 2,  4,  6],
       [12, 15, 18]])
```

Arithmetic operators in general create new arrays. To modify an existing array, use in-place operators such as *=, +=, ...

Numpy

Matrix product can be performed using the `dot` function, or the `@` operator (requires Python >= 3.5)

```
>>> x = numpy.array([[1,2], [3,4], [5,6]])
>>> y = numpy.array([[1,2,3], [4,5,6]])
>>> numpy.dot(x, y)
array([[ 9, 12, 15],
       [19, 26, 33],
       [29, 40, 51]])
```

Again, dimensions should match (in this case, the number of columns of x should be equal to the number of rows of y)

Numpy

Numpy arrays can be reshaped

```
>>> x = numpy.array([[1,2], [3,4], [5,6]])
>>> x
array([[1, 2],
       [3, 4],
       [5, 6]])

>>> y = x.reshape((2,3))
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

The order of the data is preserved, but the shape is changed

Numpy

We will mainly use this to create row and column vectors:

```
>>> x = numpy.arange(3)
>>> x
array([0, 1, 2])

>>> x.reshape((1, x.size))
array([[0, 1, 2]])

>>> x.reshape((x.size, 1))
array([[0],
       [1],
       [2]])
```

Method `ndarray.ravel()` allows reshaping the array to a 1-dimensional vector

```
>>> x = numpy.arange(12).reshape((2,2,3))

>>> x
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])

>>> x.ravel()
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
   10, 11])
```

Numpy

NOTE: Pay attention that 1-dimensional arrays (shape (n,)) are NOT row vectors (shape (1, n))

We will represent data mainly as column vectors, and data matrices will consist of horizontally stacked column vectors

Many ML libraries adopt the opposite convention — data points are represented as row vectors, and data matrices are made up of vertically stacked row vectors

Numpy

Column vectors:

- Natural correspondence with mathematical notation: $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$
- Some operations are slightly slower, unless we use a Fortran-style (column major) ordering

Row vectors:

- The elements of a single data-point are consecutive (C-style, row major ordering) — some operations can be slightly faster
- A data matrix can be iterated using a for loop (however, the iterated elements will be 1-dimensional arrays)

Numpy

Arrays can be transposed using the `.T` attribute

```
>>> x = numpy.arange(3).reshape((3,1))
>>> x
array([[0],
       [1],
       [2]])

>>> x.T
array([[0, 1, 2]])
```

Transposition can be applied also to n-dimensional array — if interested check the documentation for the semantic

Numpy

Numpy also provides functions (and methods) to perform reduction operations (e.g. sum or product of all elements, maximum, minimum)

These functions operate over the whole array as if it were one-dimensional

```
>>> x = numpy.array([[1,2,3],[4,5,6]])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.sum()
21
>>> x.max()
6
```

Numpy

If we want to perform the operation over a specific axis, we can specify the `axis` parameter

```
>>> x = numpy.array([[1,2,3],[4,5,6]])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.sum(axis=0) # sum of rows
array([5, 7, 9])
>>> x.sum(axis=1) # sum of columns
array([ 6, 15])
```

Numpy

Numpy also provides a set of element-wise functions that can be applied to all elements of an array, creating a new array

```
>>> x = numpy.array([[1,2,3],[4,5,6]])
>>> numpy.exp(x)
array([[   2.71828183,    7.3890561 ,
   20.08553692],
       [  54.59815003,  148.4131591 ,
          403.42879349]])
>>> numpy.log(x)
array([[ 0.        ,  0.69314718,  1.09861229],
       [ 1.38629436,  1.60943791,  1.79175947]])
```

Numpy

NOTE: Whenever possible, it's best to avoid iterating explicitly over array elements

The Python interpreter is slow, and loops are expensive

Numpy functions are implemented directly in C, and therefore loops are executed much faster

Numpy

script1.py

```
import numpy
s = 0
N = 10000000
x = numpy.arange(N)
for i in x:
    s += i
```

script2.py

```
import numpy
N = 10000000
x = numpy.arange(N)
s = x.sum()
```

```
$> time python script1.py
real 0m3.347s

$> time python script2.py
real 0m0.127s
```

Numpy

Arrays can be sliced

For 1-dimensional arrays, this is similar to Python lists

```
>>> x = numpy.arange(5)

>>> x[1:3]
array([1, 2])

>>> x[::2]
array([0, 2, 4])

>>> x[3]
3
```

Numpy

Multidimensional arrays allow specifying a slice for each axis

Note: if we specify a single value for an axis, we get an array with one dimension less

```
>>> x = numpy.arange(15).reshape(3,5)

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

>>> x[1, 0:3]
array([5, 6, 7])
```

If we want a 1-row matrix, we have to specify also the end of the slice:

```
>>> x[1:2, 0:3]
array([[5, 6, 7]])
```

Numpy

Numpy allows for advanced indexing

In addition to slices, we can use numpy integer arrays and boolean arrays for indexing

Using complex indexing is not always straightforward, we will mainly use 1-dimensional indices — You can check on the documentation for further information on advanced indexing

We can provide an index array in place of a slice:

```
>>> idx = numpy.array([0, 0, 2])

>>> x[idx, :]
array([[ 0,  1,  2,  3,  4],
       [ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14]])

>>> x[:, idx]
array([[ 0,  0,  2],
       [ 5,  5,  7],
       [10, 10, 12]])
```

NOTE: `x[idx, jdx]` will keep the elements whose indices are the pairs of values `(idx[i], jdx[i])`

If we want to extract all rows in `idx` and all columns in `jdx`, we can either use a two-step approach, use the `ix_` function, or reshape the index arrays

```
>>> idx = numpy.array([0, 2])
>>> jdx = numpy.array([1, 3])

>>> x[idx, jdx]
array([ 1, 13])

>>> x[idx, :][:, jdx]
array([[ 1,  3],
       [11, 13]])
```

NOTE: Since `idx` and `jdx` are 1-dimensional, `x[idx, jdx]` will keep the elements whose indices are the pairs of values `(idx[i], jdx[i])`

If we want to extract all rows in `idx` and all columns in `jdx`, we can use a two-step approach, use the `ix_` function, or reshape the index arrays

```
>>> x[numpy.ix_(idx, jdx)]
array([[ 1,  3],
       [11, 13]])

>>> x[idx.reshape((2,1)), jdx.reshape((1,2))]
array([[ 1,  3],
       [11, 13]])
```

Indexing can also use boolean arrays

```
>>> xMask = x > 5
>>> xMask
array([[False, False, False, False, False],
       [False,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True]], dtype=bool)

>>> x[xMask]
array([ 6,  7,  8,  9, 10, 11, 12, 13, 14])

>>> z=numpy.array([1,0,1], dtype=numpy.bool)
>>> z
array([ True, False,  True], dtype=bool)

>>> x[z]
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14]])
```

Slices and indexing can be also used to assign values to an array

```
>>> x = numpy.zeros(6)
>>> x[::2] = 3
>>> x
array([ 3.,  0.,  3.,  0.,  3.,  0.])

>>> x[numpy.array([True, False, True, True, True, False],
    dtype=numpy.bool)] = numpy.arange(4)
>>> x
array([ 0.,  0.,  1.,  2.,  3.,  0.])
```

NOTE: `x[:]=0` modifies the elements of the array, `x=0` binds name x to value `0`

In general, slicing creates array views

A view is an array that shares its data with a different one

We can also create explicit views

Modification to a view modifies the original array

```
>>> x = numpy.arange(5)
>>> x
array([0, 1, 2, 3, 4])

>>> y = x[0:3]
>>> y[:] = 3
>>> x
array([3, 3, 3, 3, 4])
```

Advanced indexing creates copies of an array

We can also explicitly create a copy using the method `copy`, or creating a new array through `numpy.array`

If in doubt, we can check whether an array owns its data:

```
>>> x = numpy.arange(5)
>>> x.flags.owndata
True

>>> y = x[0:3]
>>> y.flags.owndata
False
```

Numpy

An important feature of numpy array is broadcasting

Broadcasting allows applying elementwise operations, such as addition and multiplication, to arrays with different shapes

Whenever arrays have different shapes:

- 1's will be prepended to the shapes of smaller arrays until all arrays have the same number of dimensions
- Axes with shape 1 are treated as if they had the same dimension as the array with largest size along the axis, and all elements were the same along the axis

Of course, numpy arrays should have the same dimensions after broadcasting

Broadcasting examples: adding the same values to each row of a 2-D array

```
>>> x = numpy.zeros((3,4))
>>> m = numpy.arange(4)
>>> x + m
array([[ 0.,  1.,  2.,  3.],
       [ 0.,  1.,  2.,  3.],
       [ 0.,  1.,  2.,  3.]])
```

Array `m` is broadcasted to a shape `(1, 4)`, and then replicated along axis 0

Broadcasting examples: adding the same values to each row of a 2-D array - alternative

```
>>> x = numpy.zeros((3,4))
>>> m = numpy.arange(4).reshape((1,4))
>>> x + m
array([[ 0.,  1.,  2.,  3.],
       [ 0.,  1.,  2.,  3.],
       [ 0.,  1.,  2.,  3.]])
```

Array `m` already has the same number of dimensions as `x`, and gets replicated along axis 0

Broadcasting examples: adding the same values to each column of a 2-D array — note that, in this case, `m` has to be a column vector

```
>>> x = numpy.zeros((3,4))
>>> m = numpy.arange(3).reshape((3,1))
>>> x + m
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.]])
```

Array `m` already has the same number of dimensions as `x`, and gets replicated along axis 1

Numpy

Broadcasting is widely used, so get familiar with it

You can find the details at
https://numpy.org/doc/stable/user/basics.broadcasting.html

Numpy arrays can be concatenated

We can use `hstack` and `vstack` to stack vectors horizontally or vertically

For 2-D arrays:

```
>>> x1 = numpy.array([[1,2,3]])
>>> x2 = numpy.array([[4,5,6]])

>>> numpy.hstack([x1, x2])
array([[1, 2, 3, 4, 5, 6]])

>>> numpy.vstack([x1, x2])
array([[1, 2, 3],
       [4, 5, 6]])
```

Numpy

`hstack` and `vstack` can be used for N-dimensional arrays

`hstack` concatenates arrays along axis 1

`vstack` concatenates arrays along axis 0

We can also use the function `concatenate` — we need to specify the axis

- `numpy.hstack(l)` is equivalent to `numpy.concatenate(l, axis = 1)`
- `numpy.vstack(l)` is equivalent to `numpy.concatenate(l, axis = 0)`

Numpy

Numpy also provides several linear algebra functions

These can be found inside `numpy.linalg`

For example, we can compute the eigenvalue decomposition of a 2-D array:

```
>>> x = numpy.arange(9).reshape(3,3)
>>> numpy.linalg.eig(x)
(array([  1.33484692e+01,  -1.34846923e+00,
    -2.48477279e-16]),
 array([[ 0.16476382,  0.79969966,  0.40824829],
        [ 0.50577448,  0.10420579, -0.81649658],
        [ 0.84678513, -0.59128809,  0.40824829]]))
```

We will discuss the different functionalities as needed

Matplotlib

In many cases it will be useful to plot functions, data points and so on

Several libraries are available, we will use `matplotlib`

The library has a huge number of functionalities, we won't go into details much

The main module we will consider is `pyplot`

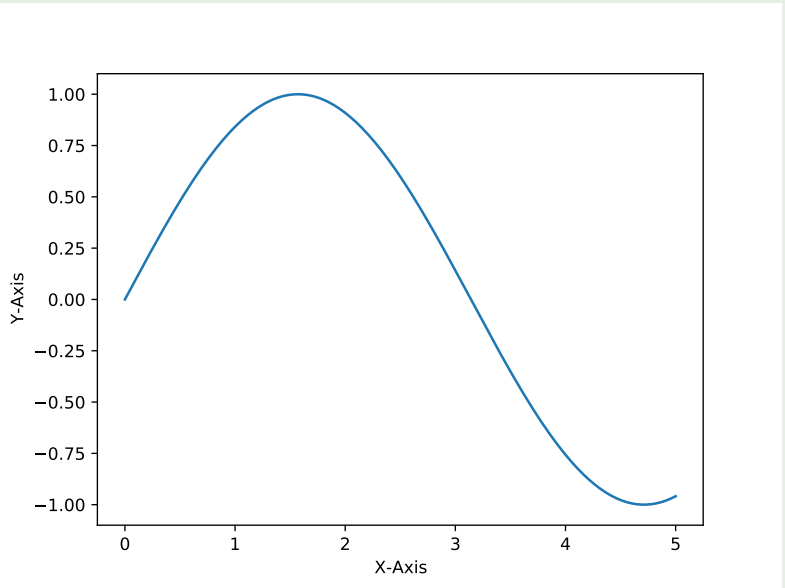
Tutorials:

- https://matplotlib.org/stable/tutorials/index.html
- https://matplotlib.org/stable/tutorials/introductory/pyplot.html#sphx-glr-tutorials-introductory-pyplot-py

Matplotlib

Plotting values can be achieved through the `plot` function

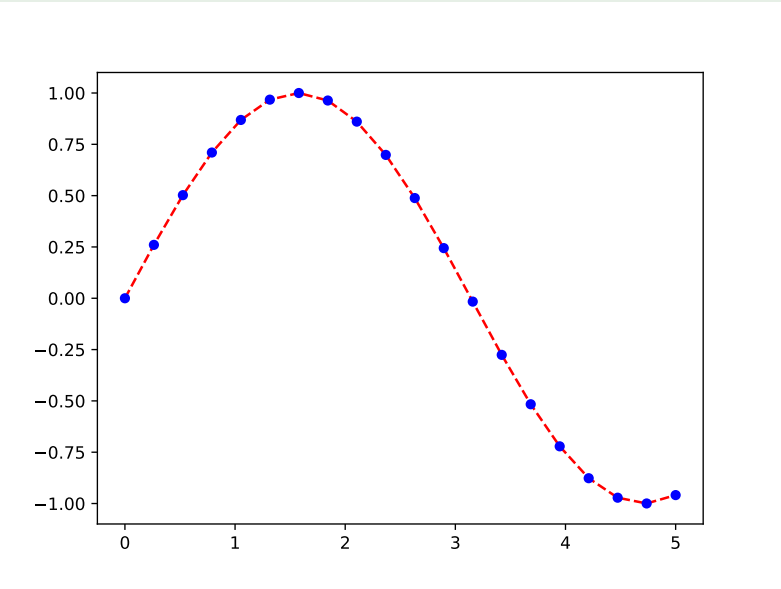
```
import matplotlib.pyplot as plt
x = numpy.linspace(0, 5, 1000)
plt.plot(x, numpy.sin(x))
plt.xlabel('X-Axis')
plt.ylabel('Y-Axis')
plt.show()
```



Matplotlib

We can specify color, line style (solid, dashed, points, ...), style attributes (e.g. line width) and so on

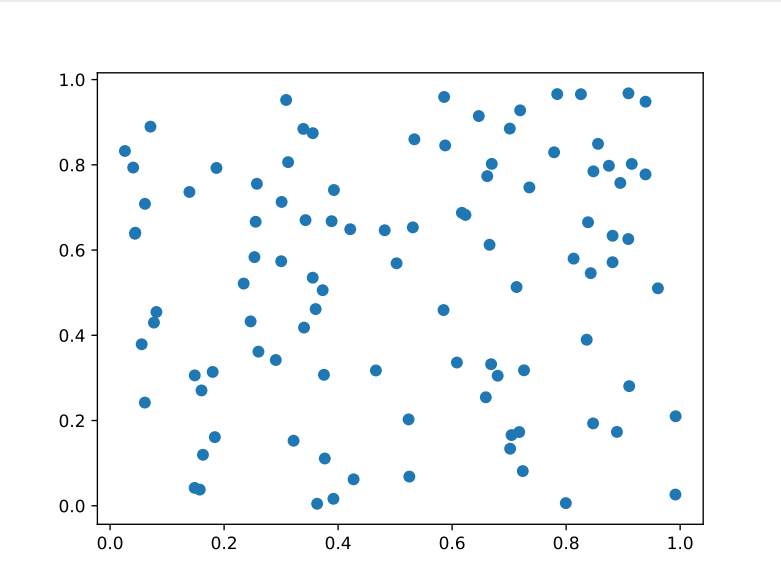
```
import matplotlib.pyplot as plt
x = numpy.linspace(0, 5, 20)
plt.plot(x, numpy.sin(x), color='r', linestyle='--')
plt.plot(x, numpy.sin(x), 'bo', markersize = 5)
plt.show()
```



Matplotlib

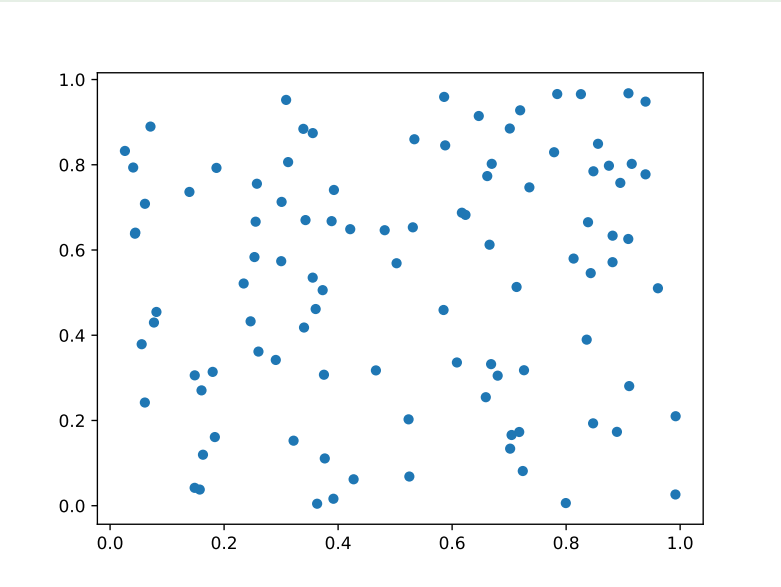
We can visualize 2-D data using scatter plots

```
import matplotlib.pyplot as plt
D = numpy.random.random((2, 100))
plt.scatter(D[0], D[1])
plt.show()
```



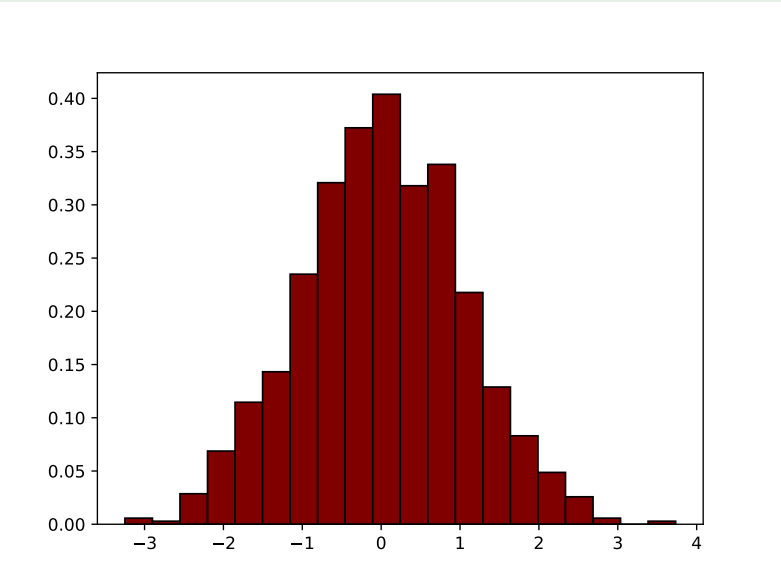
Using `plot`:

```
import matplotlib.pyplot as plt
D = numpy.random.random((2, 100))
plt.plot(D[0], D[1], linestyle='', marker='.',
    markersize=10)
plt.show()
```



Visualizing data distributions through histograms:

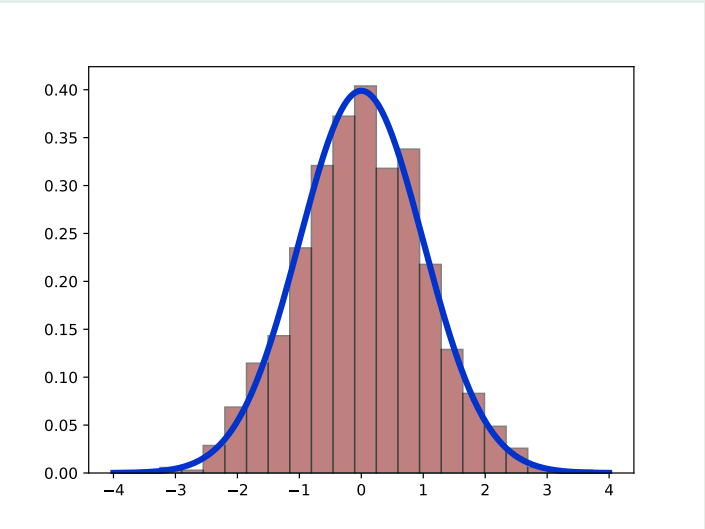
```
import matplotlib.pyplot as plt
D = numpy.random.normal(size=1000)
plt.hist(D, bins = 20, density=True, ec='black',
    color='#800000')
plt.show()
```



Matplotlib

We can add multiple plots to the same figure. `show` is used to show a picture after all elements have been added

```
import matplotlib.pyplot as plt
D = numpy.random.normal(size=1000)
plt.hist(D, bins = 20, density=True, ec='black', color='#800000',
    alpha = 0.5)
x = numpy.linspace(-4, 4, 1000)
y = 1.0/(2*numpy.pi)**0.5 * numpy.exp(-0.5 * x**2)
plt.plot(x, y, color=(0.0, 0.2, 0.8), linewidth=4)
plt.show()
```



Matplotlib

We will discuss additional functionalities as needed

Lots of examples and material are also available on the web