

# NumPy, SciPy and Visualization

## Quantitative Applications for Data Analysis

Alexey Fedoseev

March 11, 2025



# Today's class

Today we will discuss two packages that are often considered as the basis of many scientific and numerical computing tasks in python:

- NumPy - the fundamental package for scientific computing with Python, containing a powerful N-dimensional array object, and useful linear algebra, Fourier transform, and random number capabilities.
- SciPy - provides many user-friendly and efficient numerical routines such as routines for numerical integration and optimization.

To use **NumPy** and **SciPy** with Miniforge, you need to install them first:

```
conda install numpy scipy
```

## Multidimensional lists in Python

The element of a list in Python can be of any type, including a list, that is we can create a list of lists or multidimensional list. For example, this is how you can create a Vandermonde matrix:

```
>>> vander_matrix = [[1.0, 1.0, 1.0], [1.0, 2.0, 4.0], [1.0, 3.0, 9.0]]
```

Here we have a three-element list where each element consists of a three-element list.

```
>>> vander_matrix[0]
[1.0, 1.0, 1.0]
>>> vander_matrix[1]
[1.0, 2.0, 4.0]
>>> vander_matrix[0][0]
1.0
>>> vander_matrix[1][1]
2.0
```

Remember that list indices in Python start at 0.

## NumPy arrays

The NumPy array is similar to a list but where all the elements of the list are of the same type.

NumPy has a number of functions for creating arrays. The first of these, the `array` function, converts a list to an array.

```
>>> import numpy
>>> vander_matrix
[[1.0, 1.0, 1.0], [1.0, 2.0, 4.0], [1.0, 3.0, 9.0]]
>>> vander_matrix_numpy = numpy.array(vander_matrix)
>>> vander_matrix_numpy
array([[1., 1., 1.],
       [1., 2., 4.],
       [1., 3., 9.]])
```

Remember to `import numpy` module in your script.

## NumPy arrays

The second way arrays can be created is using the NumPy `linspace` function. It creates an array of  $N$  evenly spaced points between a starting point and an ending point. The form of the function is `linspace(start, stop, N)`. If the third argument  $N$  is omitted, then  $N = 50$ .

```
>>> numpy.linspace(0, 3, 7)
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. ])
```

The third way arrays can be created is using the NumPy `arange` function. The form of the function is `arange(start, stop, step)`. If the third argument is omitted `step = 1`. If the first and third arguments are omitted, then `start = 0` and `step = 1`.

```
>>> numpy.arange(0,1,0.1)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
>>> numpy.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

## NumPy arrays

A fourth way to create an array is with the `zeros` and `ones` functions which create arrays where all the elements are either zeros or ones.

```
>>> numpy.zeros(5)
array([0., 0., 0., 0., 0.])
>>> numpy.ones(5)
array([1., 1., 1., 1., 1.])
```

Very often you find that instead of typing the name of the module `numpy`, it is imported with a short alias `np`.

```
>>> import numpy as np
>>> np.ones(3)
array([1., 1., 1.])
```

## Mathematical operations with arrays

It is very easy to perform mathematical operations on every element in the array.

```
>>> vander_matrix_numpy
array([[1., 1., 1.],
       [1., 2., 4.],
       [1., 3., 9.]])
>>> vander_matrix_numpy * 2
array([[ 2.,  2.,  2.],
       [ 2.,  4.,  8.],
       [ 2.,  6., 18.]])
```

This works not only for multiplication, but for any other mathematical operation.

```
>>> vander_matrix_numpy - 1
array([[0., 0., 0.],
       [0., 1., 3.],
       [0., 2., 8.]])
```

## Mathematical operations with arrays

Multiplication of two arrays is performed element-wise.

```
>>> vander_matrix_numpy * vander_matrix_numpy  
array([[ 1.,  1.,  1.],  
       [ 1.,  4., 16.],  
       [ 1.,  9., 81.]])
```

To calculate the dot product of two arrays use function `np.dot`.

```
>>> np.dot(vander_matrix_numpy, vander_matrix_numpy)  
array([[ 3.,  6., 14.],  
       [ 7., 17., 45.],  
       [13., 34., 94.]])
```

These kinds of operations with arrays are called vectorized operations because the entire array, or “vector”, is processed as a unit. Vectorized operations are much faster than processing each element of arrays one by one.



## Multidimensional arrays

We can create a multidimensional array by applying `array` function to the multidimensional list

```
>>> numpy.array([[1,2,3,4,5],[6,7,8,9,10]])  
array([[ 1,  2,  3,  4,  5],  
       [ 6,  7,  8,  9, 10]])
```

To create a multidimensional array using the `zeros` and `ones` functions we need to specify number of rows and number of columns. In NumPy rows are always specified first.

```
>>> numpy.ones((3, 4))  
array([[1., 1., 1., 1.],  
       [1., 1., 1., 1.],  
       [1., 1., 1., 1.]])
```

Notice the way we specified the number of rows and columns: `(3, 4)`. This structure is called tuple. Tuples are very similar to lists, but the main difference between them is that the tuples cannot be changed unlike lists.

## Array indexing

NumPy offers several ways to index arrays.

```
>>> all_data = numpy.arange(10, 0, -1)
>>> all_data
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
>>> all_data[0]
10
>>> all_data[-1] # supports negative indices
1
>>> all_data[2:]
array([8, 7, 6, 5, 4, 3, 2, 1])
>>> all_data[:2]
array([10,  9])
>>> all_data[0:2] # slice items between indexes
array([10,  9])
```

While slicing between indices, the start index is included and the stop index is not included.

## Boolean indexing

Frequently we want to select or modify only the elements of an array satisfying some condition (fancy indexing).

```
>>> all_data
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
>>> (all_data <= 7) & (all_data >= 5)
array([False, False, False,  True,  True,  True, False, False, False, False])
>>> all_data[(all_data <= 7) & (all_data >= 5)]
array([7, 6, 5])
>>> even_nums = all_data[(all_data % 2) == 0]
>>> even_nums
array([10,  8,  6,  4,  2])
```

The "%" symbol is the modulo operator.

## Multidimensional slices

You can slice multidimensional arrays in a similar way.

```
>>> vander_matrix_numpy
array([[1., 1., 1.],
       [1., 2., 4.],
       [1., 3., 9.]])
>>> vander_matrix_numpy[1,1]
2.0
>>> vander_matrix_numpy[2,:]
array([1., 3., 9.])
>>> vander_matrix_numpy[1:,1:]
array([[2., 4.],
       [3., 9.]])
```

## Shape and reshape

The shape property returns a tuple of array's dimensions and can be used to change the dimensions of an array.

```
>>> seq_array = numpy.arange(1,11)
>>> seq_array
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> seq_array.shape
(10,)
```

Here the shape (10,) means the array is indexed by a single index which runs from 0 to 9.

NumPy allows you to modify the shape of an array once it already exists. The reshape function gives a new shape to an array without changing the data.

```
>>> seq_array2d = seq_array.reshape((2,5))
>>> seq_array2d
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

## Shape and reshape

Reshaping array doesn't change the data in the memory. Instead, it creates a new view that describes a different way to interpret the data.

The shape of an multidimensional array is a tuple of its dimensions where first element of the tuple represents the number of rows and the second is the number of columns.

```
>>> seq_array2d
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
>>> seq_array2d.shape
(2, 5)
```

## Shape and reshape

Specifying -1 as one of the dimensions while reshaping, forces NumPy to calculate this dimension based on the total amount of elements in the array and already specified dimensions.

```
>>> seq_array2d.reshape((5,-1))
```

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

```
>>> seq_array2d.reshape((3,-1))
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
ValueError: cannot reshape array of size 10 into shape (3,newaxis)
```

```
>>> numpy.arange(9).reshape((-1,3))
```

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

## The linalg submodule

The linalg submodule of SciPy contains useful functions for matrix algebra.

- Typical matrix functions: inv, det, norm, etc.
- More advanced functions: eig, SVD, cholesky, etc.
- Both NumPy and SciPy have a linalg module. Use SciPy, because it is compiled with optimized BLAS/LAPACK support.

```
>>> import numpy
>>> import scipy
>>> from scipy import linalg
>>> A = numpy.array([[1,2,3], [3,4,5], [1,1,2]])
>>> linalg.det(A)
-2.0
>>> scipy.dot(A, linalg.inv(A))
array([[ 1.00000000e+00,  2.22044605e-16, -2.22044605e-16],
       [ 1.66533454e-16,  1.00000000e+00, -6.66133815e-16],
       [ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00]])
```



## Solving systems of equations

The solve function in the linalg module is used to solve the system of equations  $Ax = b$ .

```
>>> A
array([[1, 2, 3],
       [3, 4, 5],
       [1, 1, 2]])
>>> b = numpy.array([3, 4, 2])
>>> b
array([3, 4, 2])
>>> x = linalg.solve(A, b)
>>> x
array([-0.5, -0.5, 1.5])
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} -0.5 \\ -0.5 \\ 1.5 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

SciPy contains all of the statistical functions that you'll probably ever need.

- The `scipy.stats` module is based around the idea of the random variable type.
- A whole variety of standard distributions are available:
  - ▶ Continuous distributions: Normal, Maxwell, Cauchy, Chi-squared, Gumbel Left-scewed, Gilbrat, Nakagami, etc.
  - ▶ Discrete distributions: Poisson, Binomial, Geometric, Bernoulli, etc.
- The random variables have all of the statistical properties of the distributions built into them already: cdf, pdf, mean, variance, moments, etc.

# Statistics

Let us create a normally distributed random variable with the mean of 1.0 and the standard deviation of 0.5.

```
>>> from scipy import stats
>>> x = stats.norm(1, 0.5)
>>> x.mean()
1.0
>>> x.median()
1.0
>>> x.std()
0.5
>>> x.var()
0.25
```

# Statistics

We can evaluate the probability distribution function, the cumulative distribution function, etc., using the pdf, cdf, etc. These functions could take a value, or an array of values, where the function will be evaluated.

```
>>> x.pdf([0, 1, 2])  
array([0.10798193, 0.79788456, 0.10798193])  
>>> x.cdf([0, 1, 2])  
array([0.02275013, 0.5, 0.97724987])
```

The `interval` method can be used to compute the lower and upper values of `x` such that a given percentage of the probability distribution falls within the interval (lower, upper). This method is useful for computing confidence intervals

```
>>> x.interval(0.95)  
(0.020018007729972975, 1.979981992270027)
```

# Visualization

There are a number of high-quality visualization packages available in Python

- `matplotlib` focuses on generating publication-quality plots
- `seaborn` targets statistical data analysis
- `ggplot` is based on the famous R package
- `Plotly` and `Bokeh` focus on interactivity
- and others

# Installing Matplotlib

To install matplotlib package run the following command in your terminal

```
$ pip install matplotlib
```

If you are using **Miniforge** you can also install matplotlib using conda package manager:

```
$ conda install matplotlib
```

matplotlib is imported using the following command

```
>>> import matplotlib.pyplot as plt
```

Also import numpy as it is frequently used together with matplotlib

```
>>> import numpy as np
```

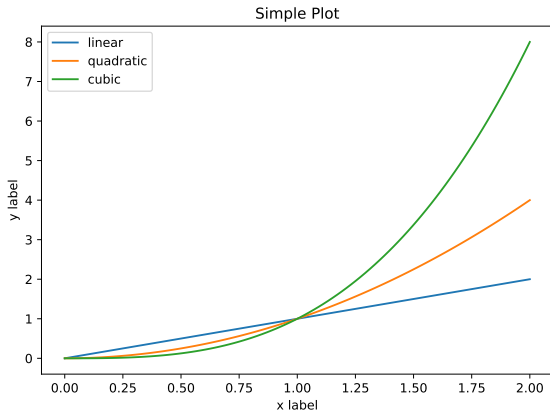
# Simple plot in matplotlib

```
import matplotlib.pyplot as plt
import numpy as np

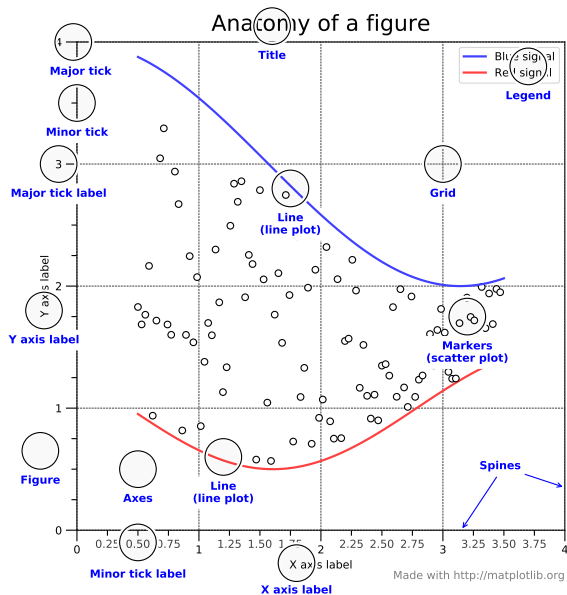
x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()

plt.show()
```



To save your plot use the command:  
`plt.savefig(filename)`

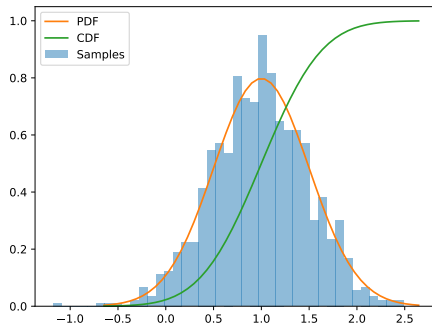




# Statistics

We can use matplotlib to visualize the distributions. In the histogram we use `density=True` to display a probability density, i.e., the area (or integral) under the histogram will sum to 1.

```
import numpy
from scipy import stats
import matplotlib.pyplot as plt
x = stats.norm(1, 0.5)
conf_interval = x.interval(0.999)
x_conf = numpy.linspace(
    conf_interval[0], conf_interval[1])
plt.hist(x.rvs(size=1000), density=True,
        bins=41, alpha=0.5, label="Samples")
plt.plot(x_conf, x.pdf(x_conf), label="PDF")
plt.plot(x_conf, x.cdf(x_conf), label="CDF")
plt.legend()
plt.show()
```



# References

NumPy reference: <https://docs.scipy.org/doc/numpy/>

SciPy reference: <https://docs.scipy.org/doc/scipy/reference/>

[https://s3.amazonaws.com/assets.datacamp.com/blog\\_assets/Numpy\\_Python\\_Cheat\\_Sheet.pdf](https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf)

Robert Johansson, Numerical Python: A Practical Techniques Approach for Industry, Apress, New York, 2015

David J. Pine, Introduction to Python for Science and Engineering, Taylor & Francis Group, 2018