Note 1

* In this plot, I will demonstrate how to generate random numbers from the Gaussian, binomial, and Poisson distributions using Python.
* Also, we will examine the distribution of the generated random numbers, using the "histogram" function, and compare them with the exact results

Note 2

* The 1st command "seed", is used to initialize the seed of the random number.
* This specifies the initial state of the generator.
* The 2nd command "rand", generates uniform random numbers between 0 and 1 as a multi-dimensional array of the specified shape.
* The 3rd command "randn", works like "rand" but draws the numbers from a normal distribution.
* The 4th and 5th commands, "binomial" and "Poisson", generate random numbers from the binomial and Poisson distributions.
* The 6th command "choice", generates random samples from the specified set.
* We will use this function to perform a random walk simulation later.
* Gaussian and uniform random numbers can be generated also by "normal" and "uniform" functions, but we will use the "randn" and "rand" functions instead.

Note 3

* First we import the common libraries for today's lesson.
* "math" library contains mathematical functions defined by the C standard, such as the "factorial" function.

Note 4

* Let us first generate random numbers from a Gaussian distribution.
* In the code example shown here, we set the average and the standard deviation to be zero and one, respectively.
* With this choice, the distribution is called the standard normal distribution.
* Here we generate one hundred thousand random numbers using an initial seed 0.
* The 5th line is the main operation of this example, where an array of "N" random numbers is generated from a Gaussian distribution with the given average and standard deviation, and stored as an array X.
* From the 6th line to the end, we have the code used to plot the random numbers.
* If you run the code example, you will see the "N" red points distributed around zero, mostly between -2 and +2.

Note 5

* Next, let us examine the distribution of random numbers we have just generated in the previous code example, and compare it with the Gaussian distribution function shown here.
* In this code example, the 1st line calculates and plots the histogram of the random numbers stored in X using a single "histogram" command.
* The remaining part is only to plot the Gaussian distribution function Eq.(D1) and to draw the figure.
* As you can see in the figure the agreement between the two is almost perfect.

Note 6

* Next, we calculate the auto-correlation function defined in Eq.(D2) for the sequence random numbers we have generated.
* We will study the correlation function in detail next week.
* For now, what we need to know is that the correlation function gives a measure of how "related" two variables are.
* In this case, we are interested in the correlation between the random numbers.
* In other words, if I know what number just came out of the generator, what can I say about the next number that will come out?
* If the correlation is zero, as given in Eq.(D4), then knowing one random number, tells me absolutely nothing about the next random number that can come out.
* This is expected from any good random number generator.
* In this lesson, we will use the correlation function to confirm two basic properties of a random sequence.
* The 1st confirmation is on its variance, which should be equal to the square of the standard deviation, as shown in Eq.(D3).
* The 2nd confirmation is on the independency of the random numbers, as given by Eq.(D3).
* In the code example shown here, we first define the "auto\_correlate" function, which calculates Eq.(D3), using the "numpy" library's own "correlate" function.
* If you run this code example, you can see that the variance of the random numbers does in fact agree with the theoretical value, which is equal to the square of the standard deviation.
* Also, we see that the correlation between distinct random numbers is zero, as shown in the figure. This assures us that the different random numbers are independent of each other.
* Stochastic variables satisfying this property are referred to as "white noise".

Note 7

* Now, let us perform the same experiment, but this time drawing the random numbers from a binomial distribution.
* As we have seen, this describes the distribution of the results for an experiment with two possible outcomes.
* The typical example is that of a coin toss.
* On every toss, we can obtain heads with probability p, and tails with probability (1-p).
* If the coin is fair, then we of course expect p = 1-p = 0.5.
* This is the case we will consider.
* Here, we are interested in looking at the number of head or success results after one hundred coin tosses.
* To obtain reliable statistics, we perform this experiment one hundred thousand times, and each time we count the number of times head appears.
* The code shown here generates the results of the experiment, "number of heads after 100 coin tosses", for 100,000 trials, using the single command "binomial".
* The results are then stored in X and plotted.
* As you can see, all the results are scattered around an average of 50, and are almost all contained within the range between 30 and 70.

Note 8

* We again compare the distribution with the expected theoretical result.
* The histogram is calculated and plotted by calling "hist", and then we compute the exact distribution with the "binomial" function.
* As you can see in the figure, the agreement between the two is again good.

Note 9

* Now let us check the correlation between the random numbers.
* We use the same auto\_correlate function as before.
* To compare with the theory, we compute the exact value for i=0, which is given by Eq. (D9), and should be equal to σ2.
* In this case σ2=25.
* This is exactly what we observe when we plot our data.
* In addition, we see that there is no correlation between different random numbers.

Note 10

* Finally, let us perform the experiment using the Poisson distribution.
* As discussed before, this can be used to describe the probability of n "events" occurring within some time interval, assuming the expected value is a.
* The typical example would be the number of requests received by a website within a one hour time period.
* Here, we set the expected value, or average value to be a=10, and we generate one hundred thousand random numbers.
* As before, this can all be done in a single line, using the "numpy" library functions.
* The results are stored in X and plotted.
* As you can see, all the results are scattered around an average of 10, and fall within the interval between 0 and 20.

Note 11

* Again, we compare the distribution with the expected theoretical result.
* As you can see, the agreement is very good.