

Tools in Scientific Computing
Prof. Aditya Bandopadhyay
Department of Mechanical Engineering
Indian Institute of Technology, Kharagpur

Lecture – 20
Probability density functions and sampling

(Refer Slide Time: 00:35)

The image shows a Jupyter Notebook interface with a hand-drawn diagram. The diagram compares Monte-Carlo and Dynamical simulation. It features a bell-shaped curve representing a probability density function $f(x)$ and a rectangular function. Handwritten notes include 'Monte-Carlo states' and 'Dynamical evolve', and mathematical expressions like 'Sampling ← distribution', $\int f(x) dx = 1$, and $\sum \frac{1}{6} = 1$.

Hi everyone to this new lecture, in this lecture we are going to look at random numbers. In particular we are going to look at a Monte Carlo simulation and we will juxtapose against dynamical simulation and that will be covered in the next lecture, but for now Monte Carlo simulation is a way of generating states.

So, different states of a system and then finding out the statistics of those states, whereas, a dynamical simulation is something in which you evolve a certain state over time. You do not randomly generate a state, but you let states evolve, but the point is in each of these kinds of simulation you need to sample various random numbers ok. So, sampling happens from a distribution.

So, imagine you have a probability distribution function. So, I will tell you what it is. So, pdf which looks something like this, alright. So, the pdf if we call it $f(x)$, so, it must have certain properties and the most important property is that $\int_{x_0}^{x_{max}} f(x) dx$ must be equal to 1, so the integration $\int_{x_0}^{x_m} f(x) dx = 1$. So, x is the random variable, $f(x)$

is the probability density function of the random variable x and the integral from x_{\min} to x_{\max} must be 1 that is the total probability of all the variables x must be equal to 1.

And, this is quite similar to the toss of a coin where each or rather the role of a die where each face has $1/6$ probability and you have 6 faces. So, the summation of all probabilities is equal to 1. So, this is more of a discrete probability distribution whereas, this is more of a continuous distribution and the way to quantify it is through a probability distribution function or a probability density function.

And, so, the higher value over here means these variables are more likely to occur than these variables. Wherever the pdf is low or the value of the pdf is low you have a lower probability of sampling that number. So, uniform distribution something like this meaning that all the variables have equal probability of being sampled, ok.

So, suppose this goes from 0 to 6, right; so, the uniform distribution can you tell me what form it must have? So, $\int_0^6 f(x)dx = 1$ because $f(x)$ is equal to constant(c), so, this is simply $6c = 1$. So, the constant is $c = 1/6$. So, the probability density function in the case of a uniform distribution is simply $1/(x_{\max} - x_{\min})$.

So, I mean this is in a nutshell. I mean as we go along in this lecture we will see various things and how they can be of various utilities, you know we will see how to approximate π using the Monte Carlo simulation, the very classic dart throwing problem. And, then we will look at a very famous problem it is called as the Buffon needle problem in which we also get an approximation to π .

(Refer Slide Time: 04:19)

```
[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive
from scipy.integrate import solve_ivp
from mayavi import mlab

[2]: def mu_effect(mu = 1):
    x = np.linspace(-3, 3, 100); y = np.linspace(-3, 3, 100);
    X, Y = np.meshgrid(x, y);
    U = mu*X**2;
    V = -Y;
    #plt.quiver(X, Y, U/(U**2+V**2)**0.5, V/(U**2+V**2)**0.5);
    plt.streamplot(X, Y, U, V, integration_direction='forward', density=1.5);
    ax = plt.gca(); ax.set_aspect(1);
    plt.axvline(0); plt.axhline(0);
    plt.plot(mu**0.5, 0, 'ok');
    plt.plot(-mu**0.5, 0, 'sr');
    w = interactive(mu_effect, mu=(-1, 1, 0.1))
    w

Error displaying widget: model not found

[3]: def snb(a = 0.4, b = 1.0):
```

So, let us begin. Let me select the Python kernel and yeah. So, straight away let me import all these things from a previous snippet.

(Refer Slide Time: 04:30)

```
[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive

[2]: N = 10000;
s = np.random.uniform(0, 6, size=(N,));

NameError                                Traceback (most recent call last)
<ipython-input-2-aa8ed3a50168> in <module>
      1 N = 10000;
----> 2 s = np.random.uniform(0, 6, size(N));

NameError: name 'size' is not defined

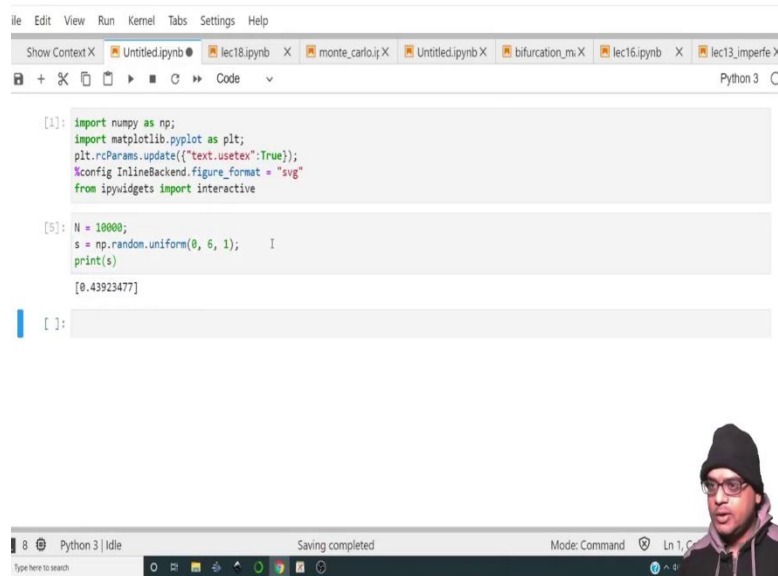
[ ]:
```

And, these are things we would require regardless of what we are trying to do, and it is just bringing in an interactive widget numpy. We will need scipy later on, but we will import it as and when we need it, alright. So, the first thing is let us find out a bunch of uniformly distributed variables. So, let me run this alright.

So, let us define the number of samples as N and let me define it as 10000, alright. Then let me define s as the samples; so s will be np.random.uniform and let us say we want to

sample between 0 and 6 and we will say we want N number of picks ok. So, this will give us no sorry, this has to be size equal to this alright.

(Refer Slide Time: 05:46)



```
file Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m X lec16.ipynb X lec13_imperfe X
Python 3

[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive

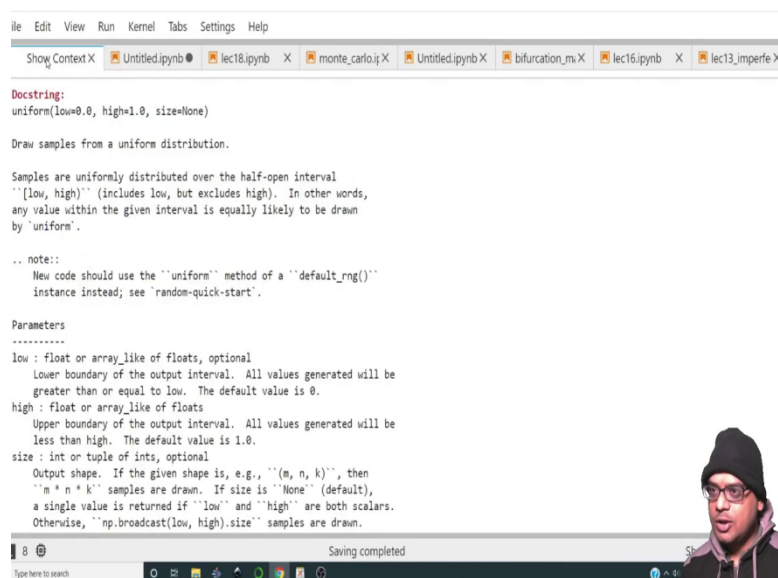
[5]: N = 10000;
s = np.random.uniform(0, 6, 1);
print(s)

[0.43923477]

[ ]:
```

So, now what we have done is we have sampled 10000 variables from a uniform distribution with a minimum value is 0 and a maximum value is 6.

(Refer Slide Time: 06:01)



```
file Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m X lec16.ipynb X lec13_imperfe X

Docstring:
uniform(low=0.0, high=1.0, size=None)

Draw samples from a uniform distribution.

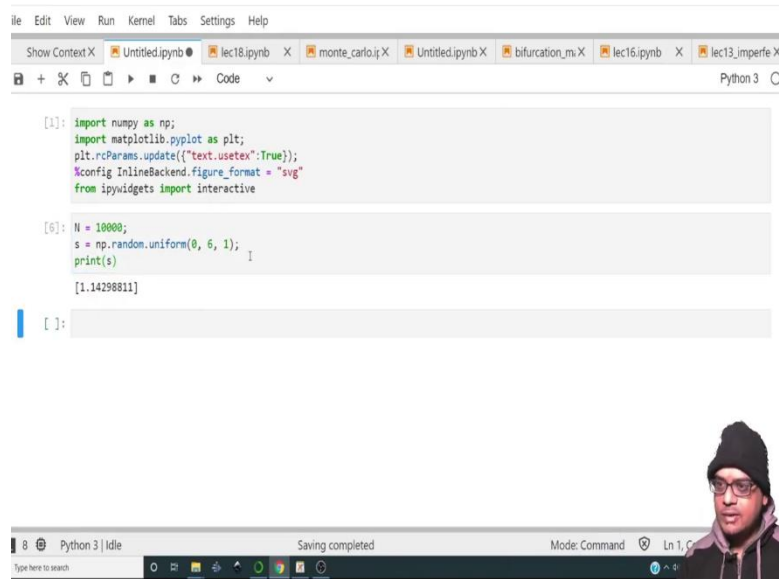
Samples are uniformly distributed over the half-open interval
"[low, high)" (includes low, but excludes high). In other words,
any value within the given interval is equally likely to be drawn
by "uniform".

.. note::
    New code should use the "uniform" method of a "default_rng()"
    instance instead; see "random-quick-start".

Parameters
-----
low : float or array_like of floats, optional
    Lower boundary of the output interval. All values generated will be
    greater than or equal to low. The default value is 0.
high : float or array_like of floats
    Upper boundary of the output interval. All values generated will be
    less than high. The default value is 1.0.
size : int or tuple of ints, optional
    Output shape. If the given shape is, e.g., "(m, n, k)", then
    "m * n * k" samples are drawn. If size is "None" (default),
    a single value is returned if "low" and "high" are both scalars.
    Otherwise, "np.broadcast(low, high).size" samples are drawn.
```

So, let me just show you the contextual help for this. So, uniform, low, high, size this is what we need. In fact, if I give this 1, I will get only one value let me print it out.

(Refer Slide Time: 06:18)



```
File Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m... X lec16.ipynb X lec13_imperfe... X
Python 3

[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({'text.usetex': True});
%config InlineBackend.figure_format = 'svg'
from ipywidgets import interactive

[6]: N = 10000;
s = np.random.uniform(0, 6, 1);
print(s)

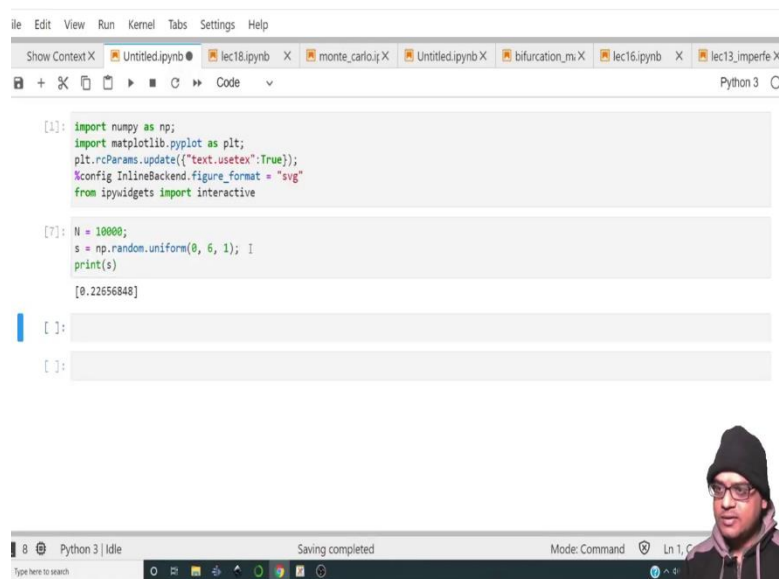
[1.14298811]

[ ]:

Python 3 | Idle Saving completed Mode: Command Ln 1, C
```

So, this is one random value. If I run this snippet again I will get another random value.

(Refer Slide Time: 06:20)



```
File Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m... X lec16.ipynb X lec13_imperfe... X
Python 3

[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({'text.usetex': True});
%config InlineBackend.figure_format = 'svg'
from ipywidgets import interactive

[7]: N = 10000;
s = np.random.uniform(0, 6, 1);
print(s)

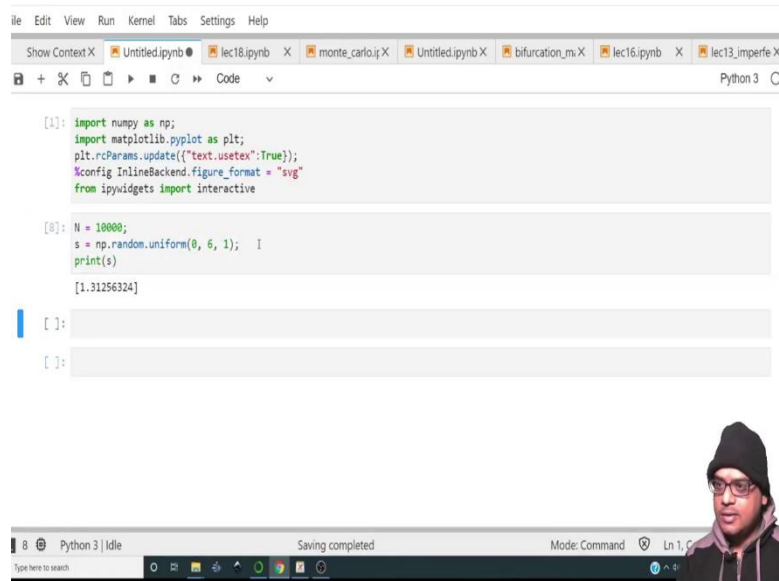
[0.22656848]

[ ]:

[ ]:

Python 3 | Idle Saving completed Mode: Command Ln 1, C
```

(Refer Slide Time: 06:21)



```
File Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m... X lec16.ipynb X lec13_imperfe... X
Python 3

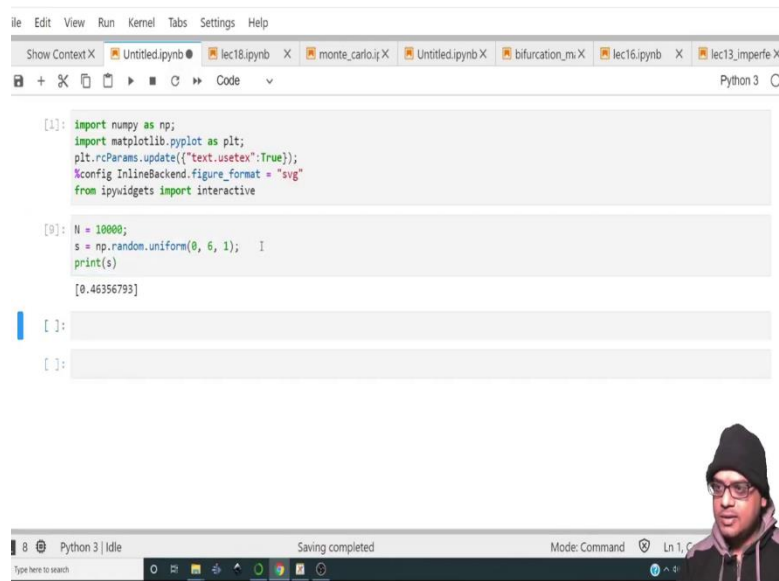
[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive

[0]: N = 10000;
s = np.random.uniform(0, 6, 1); I
print(s)
[1.31256324]

[ ]:
[ ]:
```

Python 3 | Idle Saving completed Mode: Command Ln 1, C

(Refer Slide Time: 06:23)



```
File Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m... X lec16.ipynb X lec13_imperfe... X
Python 3

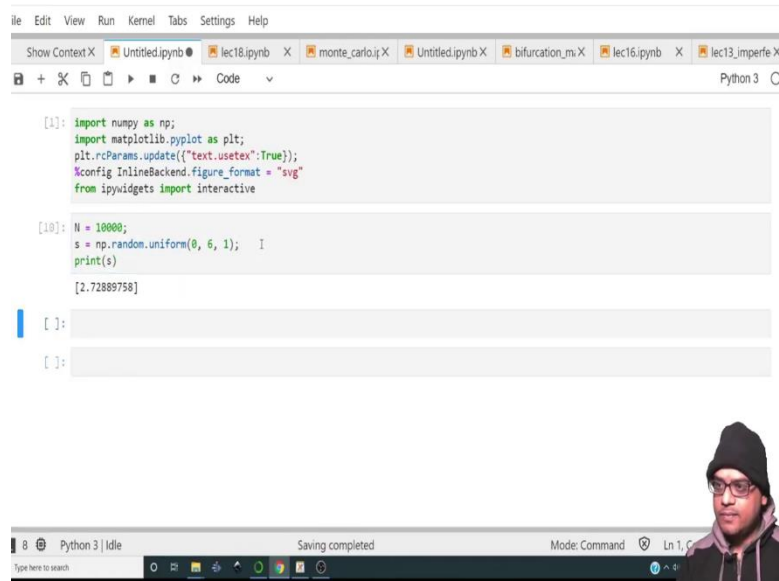
[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive

[0]: N = 10000;
s = np.random.uniform(0, 6, 1); I
print(s)
[0.46356793]

[ ]:
[ ]:
```

Python 3 | Idle Saving completed Mode: Command Ln 1, C

(Refer Slide Time: 06:22)



```
File Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m... X lec16.ipynb X lec13_imperfe... X
Python 3

[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive

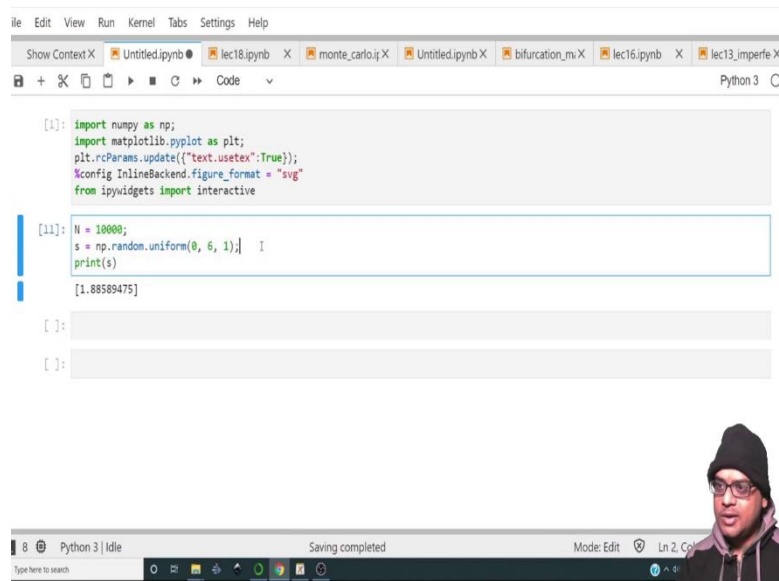
[10]: N = 10000;
s = np.random.uniform(0, 6, 1); I
print(s)

[2.72889758]

[ ]:
[ ]:
```

Python 3 | Idle Saving completed Mode: Command Ln 1, C

(Refer Slide Time: 06:24)



```
File Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m... X lec16.ipynb X lec13_imperfe... X
Python 3

[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive

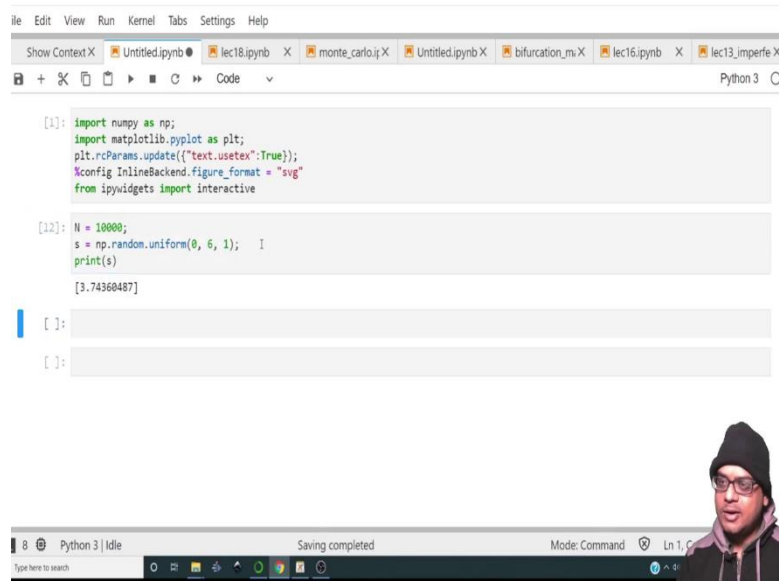
[11]: N = 10000;
s = np.random.uniform(0, 6, 1); I
print(s)

[1.88589475]

[ ]:
[ ]:
```

Python 3 | Idle Saving completed Mode: Edit Ln 2, C

(Refer Slide Time: 06:24)




```
file Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m_X lec16.ipynb X lec13_imperfe X
Python 3

[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({'text.usetex':True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive

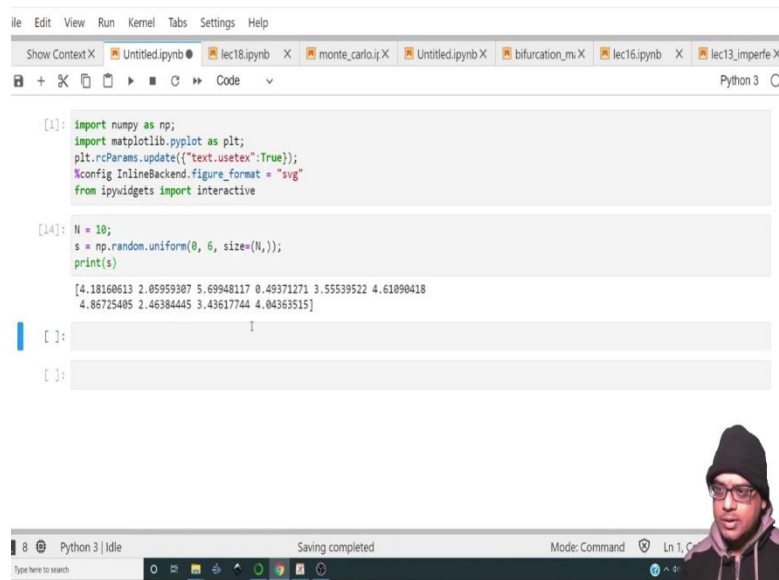
[2]: N = 10000;
s = np.random.uniform(0, 6, 1); I
print(s)
[3.74360487]

[ ]:
[ ]:
```



If I run this again; run this again, see if I keep running it I will get a host of random values uniformly sampled ok.

(Refer Slide Time: 06:31)




```
file Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m_X lec16.ipynb X lec13_imperfe X
Python 3

[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({'text.usetex':True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive

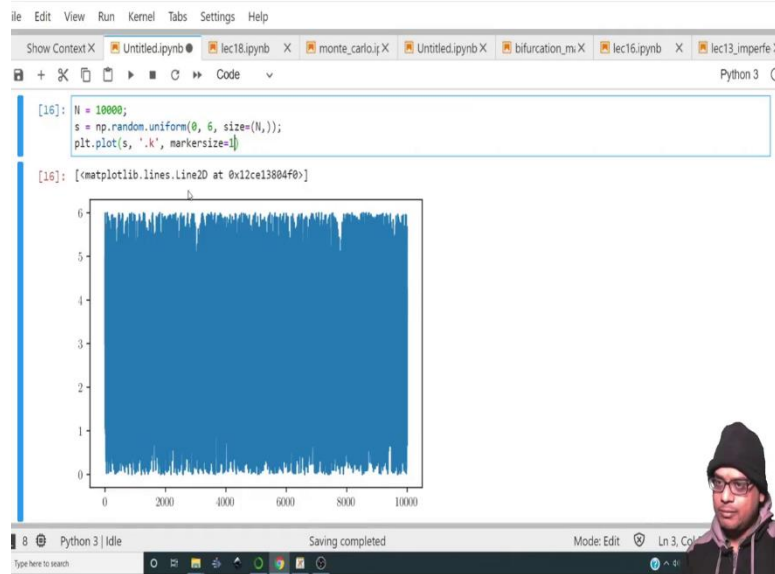
[4]: N = 10;
s = np.random.uniform(0, 6, size=(N,));
print(s)
[4.18160613 2.05959307 5.69948117 0.49371271 3.55539522 4.61090418
4.86725405 2.46384445 3.43617744 4.04363515]

[ ]:
[ ]:
```



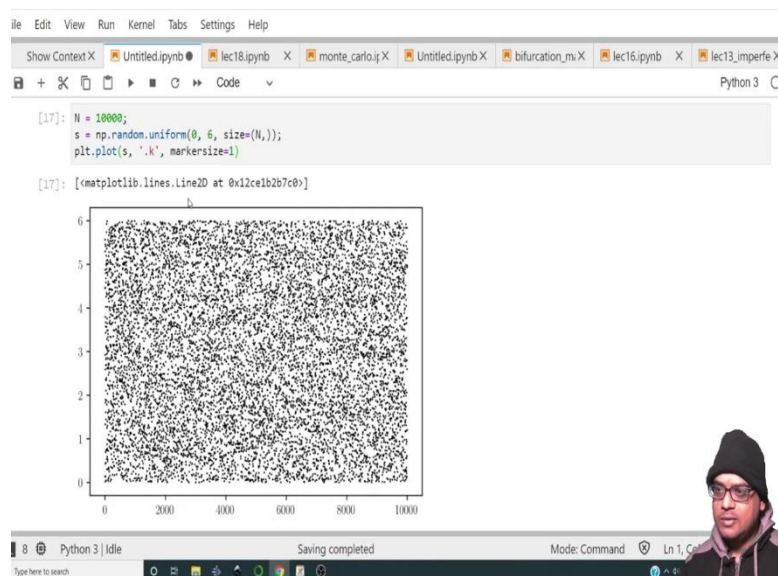
So, I will keep it like this maybe let me show 10 values. So, these are 10 values which have been sampled, alright.

(Refer Slide Time: 06:47)



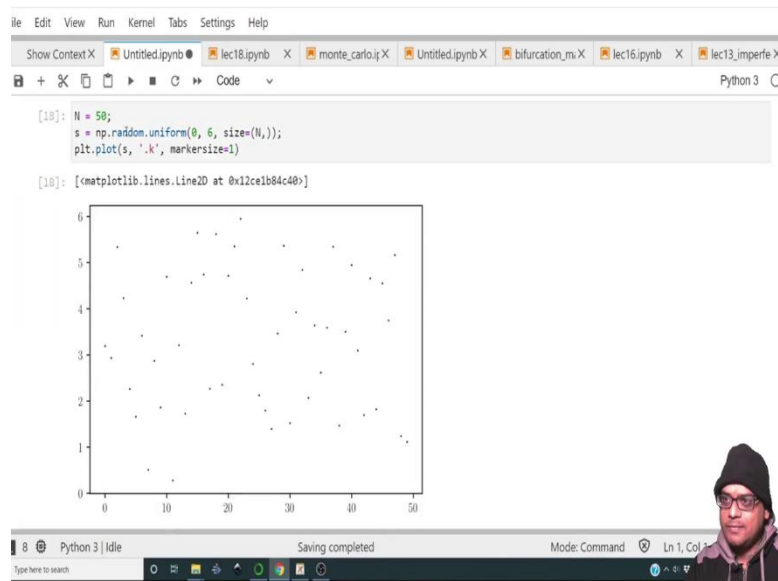
So, let me also show a plot of it. Now, let me do `plt.plot(s)`. So, now what we have is this is the array index of `s` on the x-axis. On the y-axis we have the value and it sort of fills the entire space from 0 to 6 because we have a lot of sampling going on and it will appear quite filled if you increase this further it will. So, ideally we should only plot the points rather than joining them by lines. So, let me do this. Let me write `marker size equal to 1` ok.

(Refer Slide Time: 07:32)



So, these are all the various samples. So, at a certain x value you have only one sample, ok. Do not be confused, do not think that you have multiple samples for a given x value, alright and this might be clearer if we reduce this to 50 ok.

(Refer Slide Time: 07:51)



So, for each x you have one sample and this is how you sample a lot of numbers between 0 and 6 uniformly. Now, what does the `np.random` offer us, let us see.

(Refer Slide Time: 08:04)

The screenshot shows a Jupyter Notebook interface displaying the documentation for the `np.random` module. The text in the notebook is as follows:

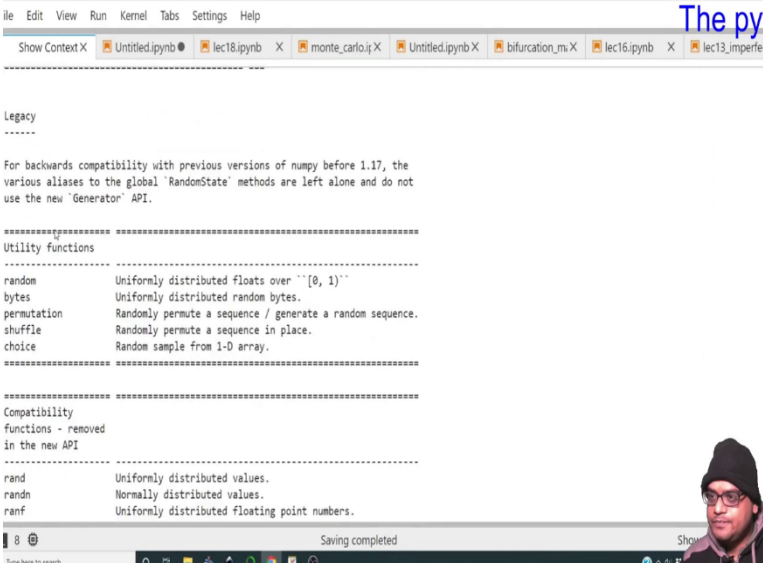
```
Type: module
String form: <module 'numpy.random' from 'F:\anaconda\lib\site-packages\numpy\random\_init_.py'>
File: f:\anaconda\lib\site-packages\numpy\random\_init_.py
Docstring:
=====
Random Number Generation
=====

Use ``default_rng()`` to create a 'Generator' and call its methods.

-----
Generator
-----
Generator      Class implementing all of the random number distributions
default_rng    Default constructor for ``Generator``
=====

-----
BitGenerator Streams that work with Generator
-----
MT19937
PCG64
Philox
SFC64
=====
```

(Refer Slide Time: 08:08)



```
file Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_mn X lec16.ipynb X lec13_imperfe X

Legacy
-----

For backwards compatibility with previous versions of numpy before 1.17, the
various aliases to the global 'RandomState' methods are left alone and do not
use the new 'Generator' API.

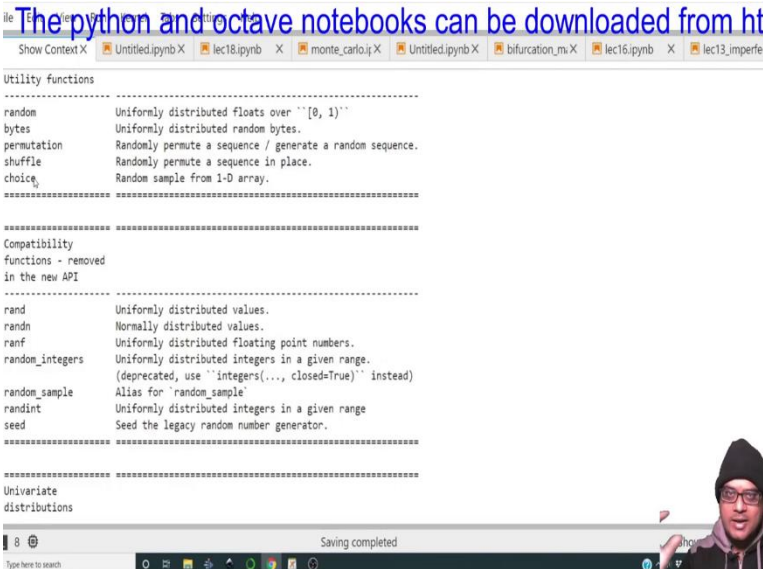
=====
Utility functions
-----
random      Uniformly distributed floats over ``[0, 1)``
bytes       Uniformly distributed random bytes.
permutation Randomly permute a sequence / generate a random sequence.
shuffle     Randomly permute a sequence in place.
choice      Random sample from 1-D array.
=====

Compatibility
functions - removed
in the new API
-----

rand        Uniformly distributed values.
randn       Normally distributed values.
randf       Uniformly distributed floating point numbers.

Saving completed
```

(Refer Slide Time: 08:17)



```
file Edit View Run Kernel Tabs Settings Help
The python and octave notebooks can be downloaded from htt
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_mn X lec16.ipynb X lec13_imperfe X

Utility functions
-----

random      Uniformly distributed floats over ``[0, 1)``
bytes       Uniformly distributed random bytes.
permutation Randomly permute a sequence / generate a random sequence.
shuffle     Randomly permute a sequence in place.
choice      Random sample from 1-D array.
=====

Compatibility
functions - removed
in the new API
-----

rand        Uniformly distributed values.
randn       Normally distributed values.
randf       Uniformly distributed floating point numbers.
random_integers Uniformly distributed integers in a given range.
(deprecated, use ``integers(..., closed=True)`` instead)
random_sample Alias for 'random_sample'
randint     Uniformly distributed integers in a given range
seed        Seed the legacy random number generator.
=====

Univariate
distributions

Saving completed
```

In the contextual help in the module we have this random which gives us random floats over 0 to 1, we have permutations, shuffling ok, choice from a random array. So, not a random array, if you have an array and we call the function choice it will select any element from that array then we have rand, randn ok.

(Refer Slide Time: 08:43)

http://www.facweb.iitkgp.ac.in/~adityab/lecture_list.html as a quick

```
Show Context X  Untitled.ipynb X  lec18.ipynb X  monte_carlo.ipynb X  Untitled.ipynb X  bifurcation_m X  lec16.ipynb X  lec13_imperfe X

Compatibility
functions - removed
in the new API
-----
rand      Uniformly distributed values.
randn     Normally distributed values.
randf     Uniformly distributed floating point numbers.
random_integers  Uniformly distributed integers in a given range.
           (deprecated, use ``integers(..., closed=True)`` instead)
random_sample  Alias for `random_sample`
randint   Uniformly distributed integers in a given range
seed     Seed the legacy random number generator.
-----

Univariate
distributions
-----
beta     Beta distribution over ``[0, 1]``.
binomial Binomial distribution.
chisquare :math:`\chi^2` distribution.
exponential  Exponential distribution.
f         F (Fisher-Snedecor) distribution.
gamma    Gamma distribution.
geometric Geometric distribution.
gumbel   Gumbel distribution.
hypergeometric  Hypergeometric distribution.
laplace  Laplace distribution.
-----

Saving completed
```



So, these are removed in the new API, but they will work nevertheless. Then we have some univariate distributions, the beta distribution, binomial distribution, chi-square distribution, exponential distribution, Fisher distribution, gamma distribution, hypergeometric distribution.

(Refer Slide Time: 08:56)

http://www.facweb.iitkgp.ac.in/~adityab/lecture_list.html as a quick reference

```
Show Context X  Untitled.ipynb X  lec18.ipynb X  monte_carlo.ipynb X  Untitled.ipynb X  bifurcation_m X  lec16.ipynb X  lec13_imperfe X

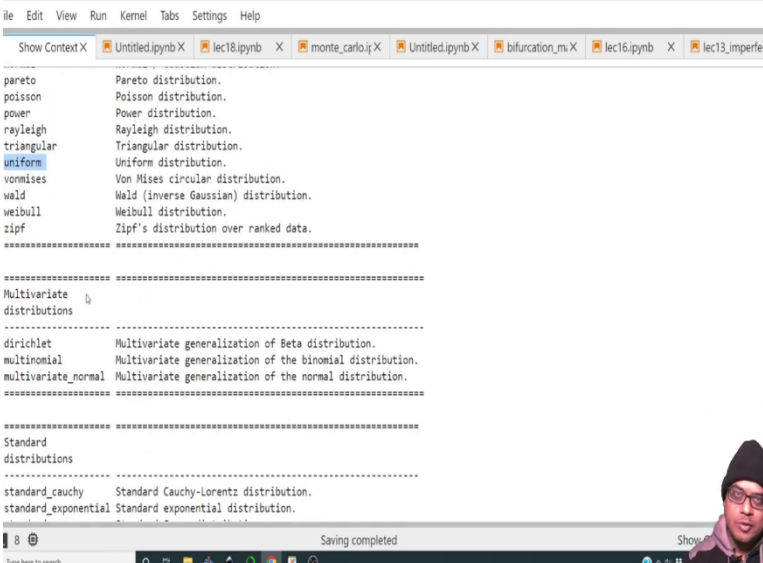
Univariate
distributions
-----
beta     Beta distribution over ``[0, 1]``.
binomial Binomial distribution.
chisquare :math:`\chi^2` distribution.
exponential  Exponential distribution.
f         F (Fisher-Snedecor) distribution.
gamma    Gamma distribution.
geometric Geometric distribution.
gumbel   Gumbel distribution.
hypergeometric  Hypergeometric distribution.
laplace  Laplace distribution.
logistic Logistic distribution.
lognormal Log-normal distribution.
logseries Logarithmic series distribution.
negative_binomial  Negative binomial distribution.
noncentral_chisquare  Non-central chi-square distribution.
noncentral_f       Non-central F distribution.
normal             Normal / Gaussian distribution.
pareto            Pareto distribution.
poisson           Poisson distribution.
power             Power distribution.
rayleigh          Rayleigh distribution.
triangular        Triangular distribution.
uniform           Uniform distribution.
vonmises          Von Mises circular distribution
-----

Saving completed
```



So, these are all various types of probability density functions, ok. We have the normal distribution which is the Gaussian you may be aware, we have the Pareto distribution, uniform distribution which we have just called, we have the Weibull distribution, Zipf distribution.

(Refer Slide Time: 09:15)



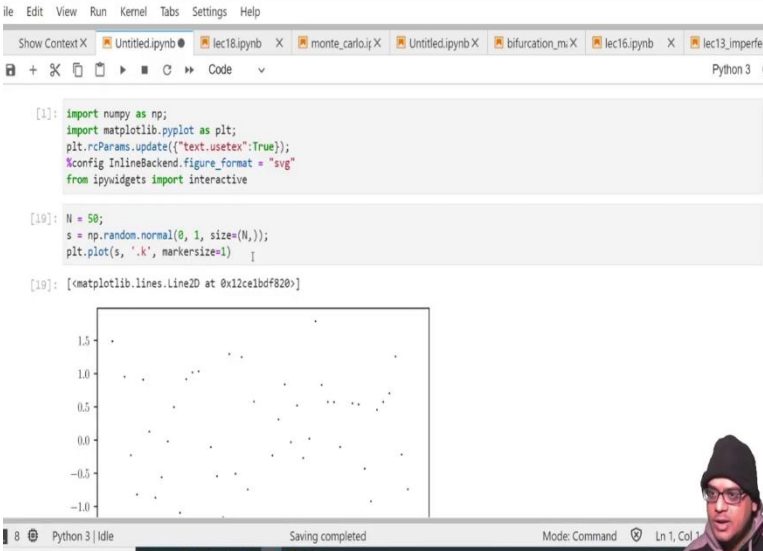
```
file Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m X lec16.ipynb X lec13_imperfe X

pareto Pareto distribution.
poisson Poisson distribution.
power Power distribution.
rayleigh Rayleigh distribution.
triangular Triangular distribution.
uniform Uniform distribution.
vonnises Von Mises circular distribution.
wald Wald (inverse Gaussian) distribution.
weibull Weibull distribution.
zipf Zipf's distribution over ranked data.
=====
Multivariate
distributions
-----
dirichlet Multivariate generalization of Beta distribution.
multinomial Multivariate generalization of the binomial distribution.
multivariate_normal Multivariate generalization of the normal distribution.
-----
Standard
distributions
-----
standard_cauchy Standard Cauchy-Lorentz distribution.
standard_exponential Standard exponential distribution.
```

Then we have multivariate distribution. So, when we go from one variable to multiple variables you have instead of a single function you have a surface, alright. So, then you have some other standard distributions and all these things.

So, with the help of this we can sort of sample various distributions sample variables from various distributions, ok. And, each distribution has its own set of parameters that you need to give. For example, in the normal distribution or the Gaussian distribution we need to supply the width of the Gaussian and the offset of the Gaussian.

(Refer Slide Time: 10:06)

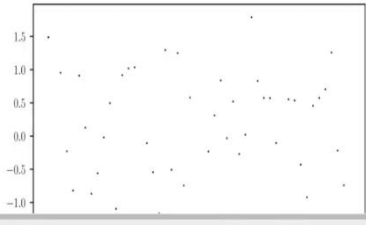


```
file Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m X lec16.ipynb X lec13_imperfe X
Python 3

[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive

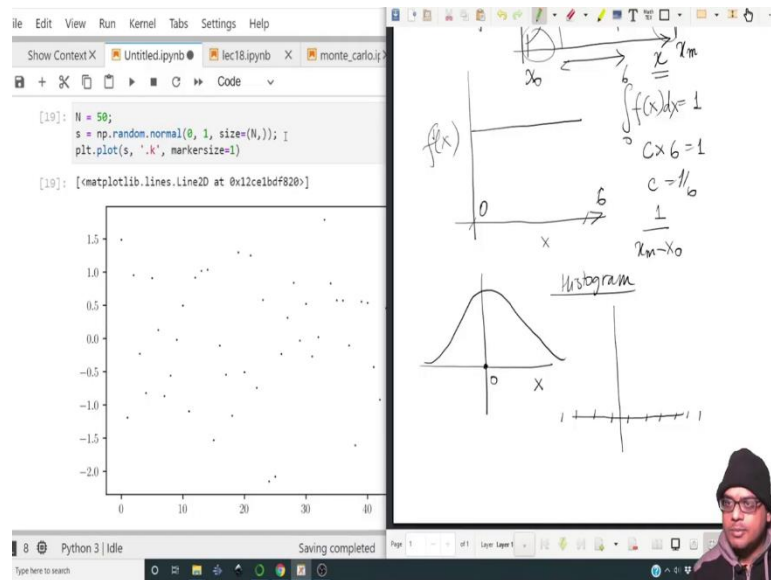
[19]: N = 50;
s = np.random.normal(0, 1, size=(N,));
plt.plot(s, '.', markersize=1)

[19]: [matplotlib.lines.Line2D at 0x12ce1bdf820]
```



So, obviously, when we call this so, let me go over here. So, now, let me go to the contextual help of this, so, the location and the scale, alright. So, let me set the location to be 0 and the scale to be 1, right. So, now, we will have a bunch of values which are picked from or something close to -2 to something close to 2.

(Refer Slide Time: 10:35)



Because the Gaussian distribution when it is centered around the origin it looks something like this ok. It is more likely to sample from the center that is the neighborhood of 0, alright.

So, in fact, what is a very convenient way of sort of sampling a lot of points and then telling someone that look the sampled points are indeed in this kind of a distribution? So, the way to do it is to do a histogram. So, histogram is sort of telling you the number of counts that you have in a certain bin. So, we can split this into various bins, alright.

(Refer Slide Time: 11:30)

The screenshot shows a Jupyter Notebook interface with the following code and output:

```
[20]: N = 50;
s = np.random.normal(0, 1, size=(N,));
print(s)
plt.plot(s, '.', markersize=1)
```

The output of the code is a list of 50 random numbers:

```
[ 0.36805134 -0.38177388 -0.46625562 -0.93755573  3.025992
 0.32773391  0.70459919 -0.3239514  -1.53423924 -1.451334
 0.02318046 -0.0740122  0.5163457  -1.2857236  -0.489063
-1.81039951  0.57153395 -0.99382358 -0.46970497  1.190411
-1.5273476  -1.61023255 -0.63027649  0.21240749  0.162141
-0.8085661  0.98980454 -0.61735186 -0.65175186 -2.043011
1.03073282  1.03891673 -1.40594398 -0.04261455 -0.721396
1.094903  -1.93406992 -1.4007028  -0.7466418  0.071885
0.16597808 -0.19691856]
```

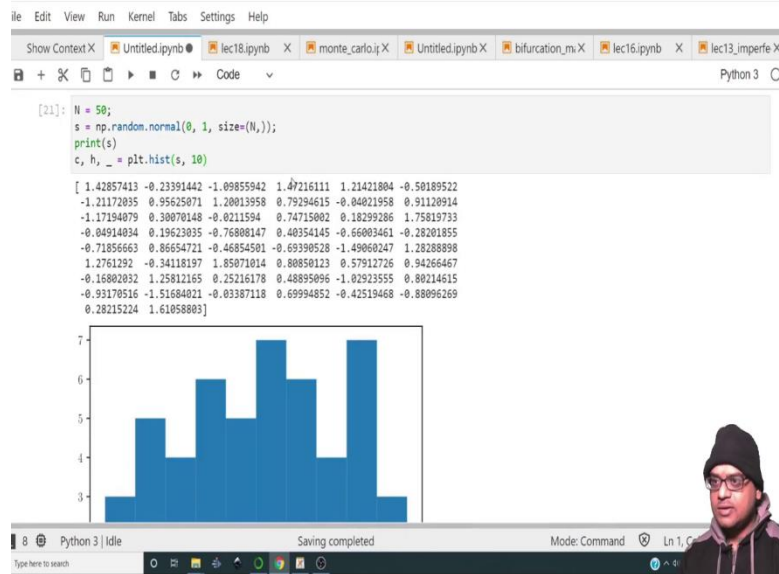
The whiteboard contains handwritten notes and diagrams:

- A graph of a function $f(x)$ with a horizontal line and a vertical axis labeled x .
- A diagram showing a point x_0 and a bin width Δx .
- Equations: $\int f(x) dx = 1$, $C \times \Delta x = 1$, $C = 1/\Delta x$, and $\frac{1}{x_m - x_0}$.
- A histogram with bars of varying heights and widths, labeled "Histogram".
- A diagram showing a point x and a bin width Δx .

And, then let me print s alright and then what we will do? Suppose this is -2, -1.7, -1.5 and so on. So, this is 0.36. So, if this is 0.3 and 0.4, in this bin we will increment the count to 1. Now what do we have? -0.38 so we have to increase the bin. So, there will be one count over here, then -0.46 and then we will increment the count somewhere over here.

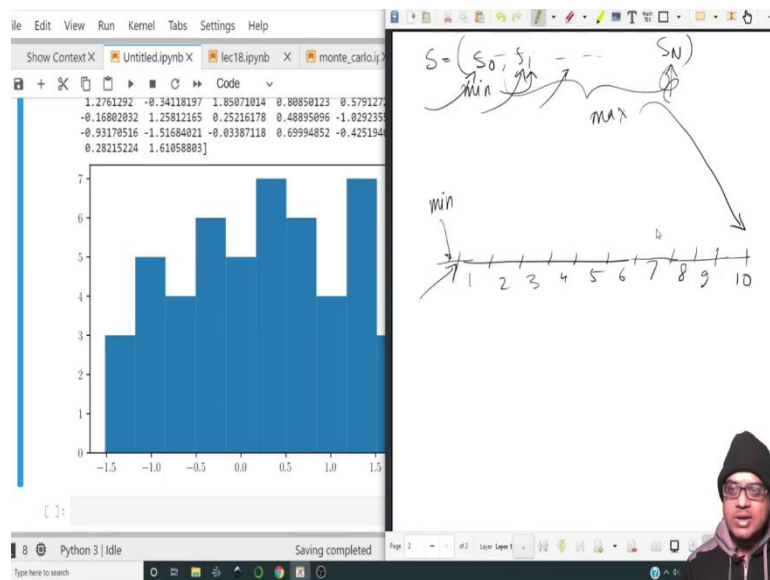
So, we will look at all the points and we will break up the domain into a lot of bins. We will count the number of occurrences in each bin and we will plot the histogram, it is something which you might have learnt in high school ok. So, because we are sampling from a distribution we expect more points or a larger count for numbers near the origin and we expect fewer counts for number away from the origin. So, let us see. Let us look at how we can do the histogram.

(Refer Slide Time: 12:55)



So, there are multiple ways of doing the histogram. In fact, there are two easy ways of doing it. So, one is directly using the matplotlib function. So, we have the count, h, patch so, but we want we do not want to access the patch. This will be equal to plt.hist, then we will pass s, then we will pass the number of bins. So, suppose I wanted over 10 bins alright. So, let me run this and show you what happens.

(Refer Slide Time: 13:21)

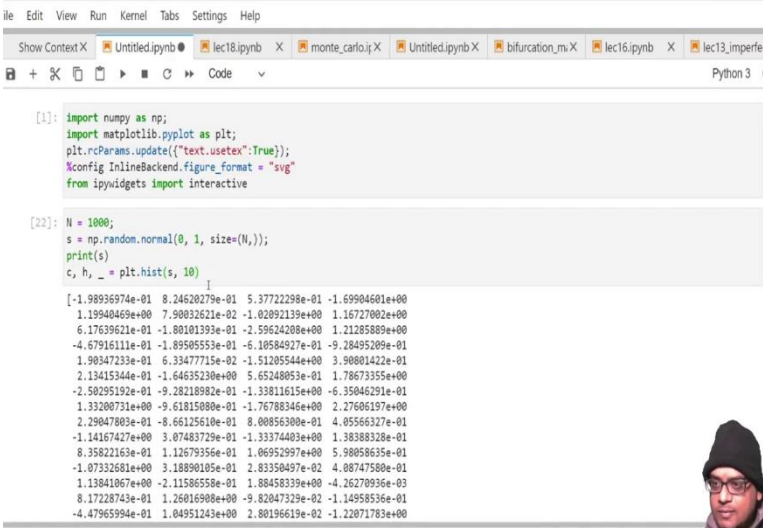


So, it has created a bin, it has created 10 number of bins. So, we have asked the function to give us the histogram over 10 number of bins. So, if you count this 1 2 3 4 5 6 7 8 9 10. So, it has split the domain into 10 bins, alright. So, 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10. So, it has split it into 10 bins and what is the edge of the bin. So, this will be the minimum value.

So, what we have is a bunch of samples $S[0]$, $S[1]$ all the way to $S[n]$. So, it will find out the minimum value amongst these picks this as the left most edge. It will find the maximum value out of all this and it will set that as the right most edge then it will split that domain into 10 bins ok. It is like taking a linspace and then it will loop over all the values $S[0]$, $S[1]$ and it will see inside which bin it lies.

So, over here we see that the bin between - 0.5 and - 0.15 or something it has 6 counts something between 0.3 or 0.2 and 0.5, it has 7 counts something between 1.2 and 1.5 it has 7 counts.

(Refer Slide Time: 15:05)



```
[1]: import numpy as np;
import matplotlib.pyplot as plt;
plt.rcParams.update({"text.usetex":True});
%config InlineBackend.figure_format = "svg"
from ipywidgets import interactive

[22]: N = 1000;
s = np.random.normal(0, 1, size=(N,));
print(s)
c, h, _ = plt.hist(s, 10)

[-1.98936974e-01  8.24620279e-01  5.37722298e-01 -1.69904601e+00
 1.19940469e+00  7.90032621e-02 -1.02092139e+00  1.16727002e+00
 6.17639621e-01 -1.80101393e-01 -2.59624208e+00  1.21285809e+00
-4.67916111e-01 -1.89505553e-01 -6.10584927e-01 -9.28495209e-01
 1.90347233e-01  6.33477715e-02 -1.51205544e+00  3.90801422e-01
 2.13415344e-01 -1.64635230e+00  5.65248053e-01  1.78673355e+00
-2.50295192e-01 -9.28218982e-01 -1.33811615e+00 -6.35046291e-01
 1.33200731e+00 -9.61815080e-01 -1.76788346e+00  2.27606197e+00
 2.29047803e-01 -8.66125610e-01  8.00856300e-01  4.05566327e-01
-1.14167427e+00  3.07483729e-01 -1.33374403e+00  1.38388328e-01
 8.35822163e-01  1.12679356e-01  1.06952997e+00  5.98058635e-01
-1.07332681e+00  3.18890105e-01  2.83350497e-02  4.08747580e-01
 1.13841067e+00 -2.11586558e-01  1.88458339e+00 -4.26270930e-03
 8.17228743e-01  1.26016908e+00 -9.82047329e-02 -1.14958536e-01
-4.47965994e-01  1.04951243e+00  2.80196619e-02 -1.22071783e+00
```

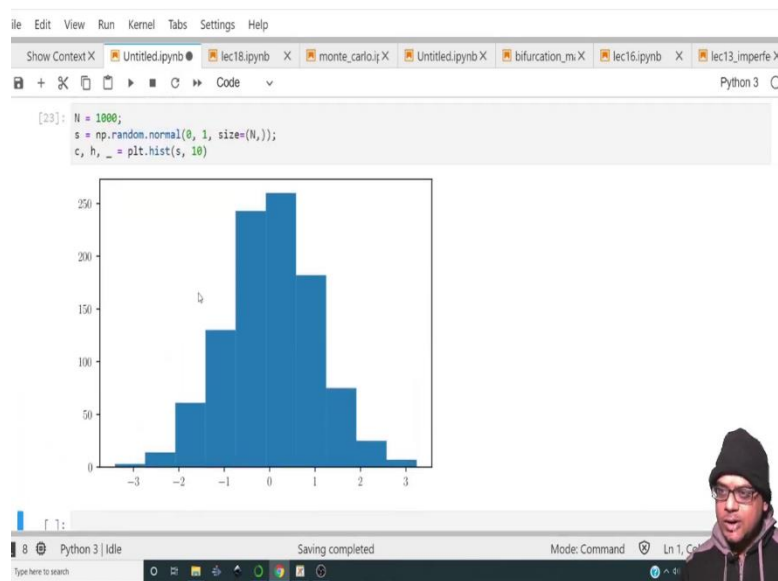
But, now let us increase the number of samples that we take. Let me make it 1000 and let me; let us run this. Let us see what.

(Refer Slide Time: 15:13)

The screenshot shows a Python IDE window with a list of 1000 random values. The values are displayed in scientific notation, ranging from approximately -1.14167427×10^0 to $3.081831487 \times 10^{-1}$. The IDE interface includes a menu bar (File, Edit, View, Run, Kernel, Tabs, Settings, Help), a toolbar, and a status bar at the bottom indicating 'Python 3 | Idle' and 'Saving completed'.

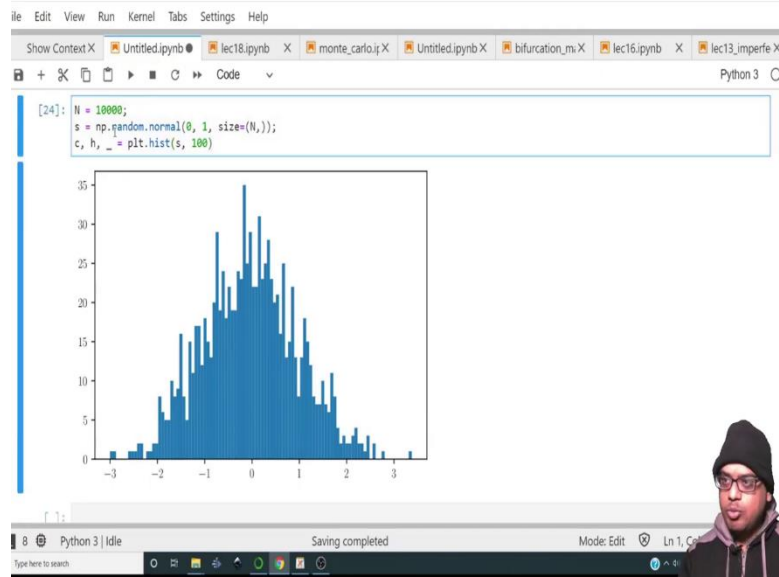
In fact, let me suppress printing because it is going to print 1000 values, alright.

(Refer Slide Time: 15:17)



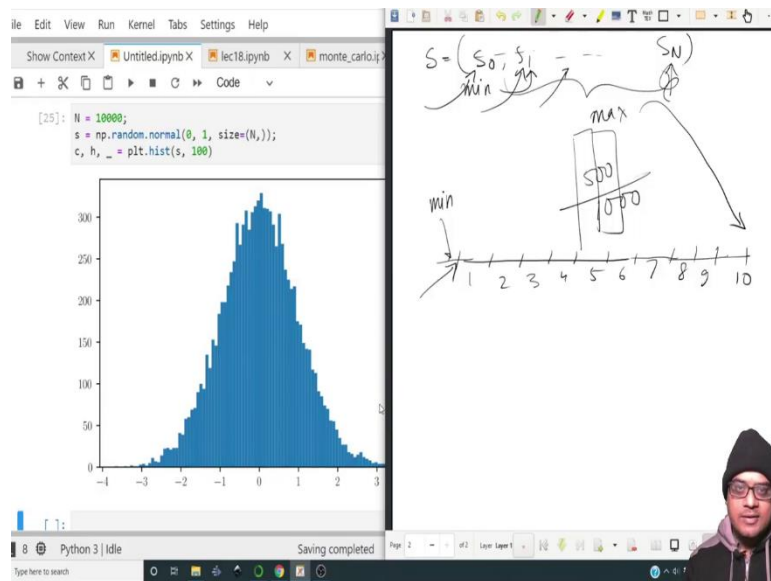
So, when we take 1000 values and we still have 10 bins, it appears to give that distribution that we desire we have more. So, we have almost 250 counts near the origin. So, we have total 500 counts near the origin, 500 out of 1000. So, we have almost half of all the samples lying near the origin and the other half lies away from the origin. Let me increase the number of bins.

(Refer Slide Time: 15:45)



Let me increase the number of bins so, you get a smoother distribution.

(Refer Slide Time: 15:51)



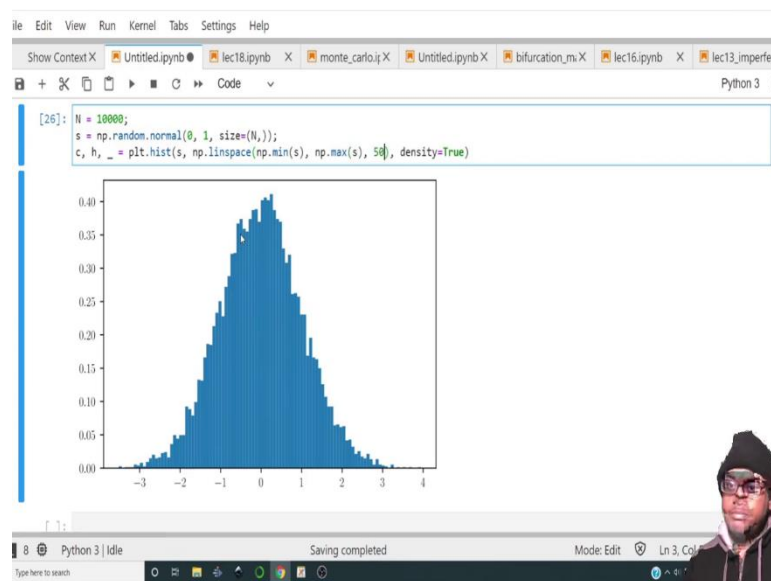
Let me increase the number of points. So, now, we do see that we have this distribution of counts, but this is the distribution of counts, it is not the probability density function. How do we plot the probability density function? We do not want the counts, but the pdf in this case.

So, essentially we are sort of representing the continuous function as a discrete function and if we divide everything by the total number; so, for example, these had 500 counts in

total, but we divide it by the number of samples, right. So, we will get the fraction of total samples that we have.

And, once you do that once you sort of see what is going on, so, the total number of all counts will be equal to the number of samples you took, ok. Once you divided by the total number you will get something which is going to be normalized.

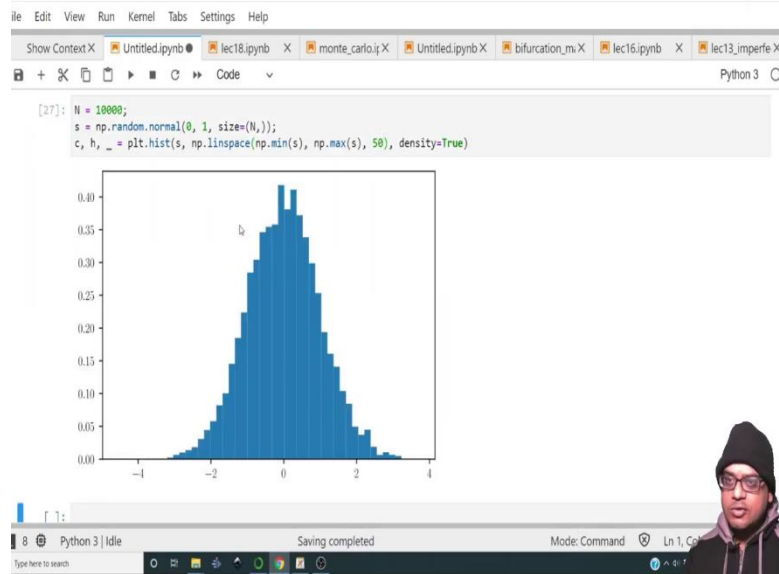
(Refer Slide Time: 16:51)



So, you go over here and you say density equal to True essentially you are dividing by the total number of samples that is N. So, now you get this distribution right. So, this particular distribution is the Gaussian, but now I have asked it to do the histogram over 100 bins alternately I could have supplied my own bins.

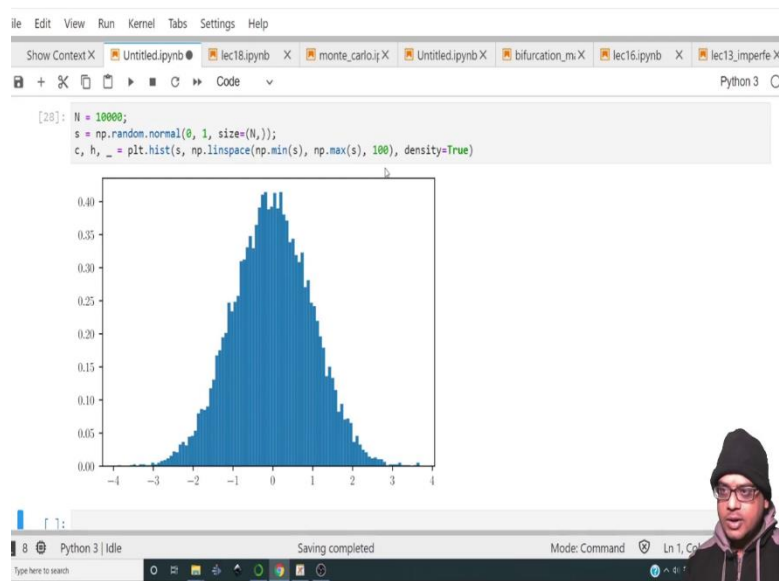
So, how do I sort of supply my own bins? So, I have already said that the bins must sort of go from the minimum value of the sample to the maximum value of the sample. So, let me do that.

(Refer Slide Time: 17:32)



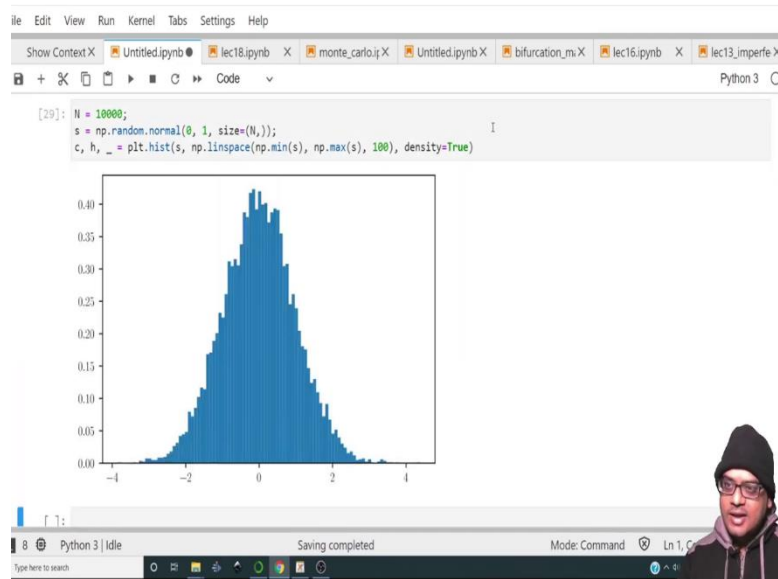
So, I will do `np.linspace(np.min(s), np.max(s),50)` and let us take 50 bins. So, now we have histogram which takes 50 bins. Let me do 100.

(Refer Slide Time: 17:54)



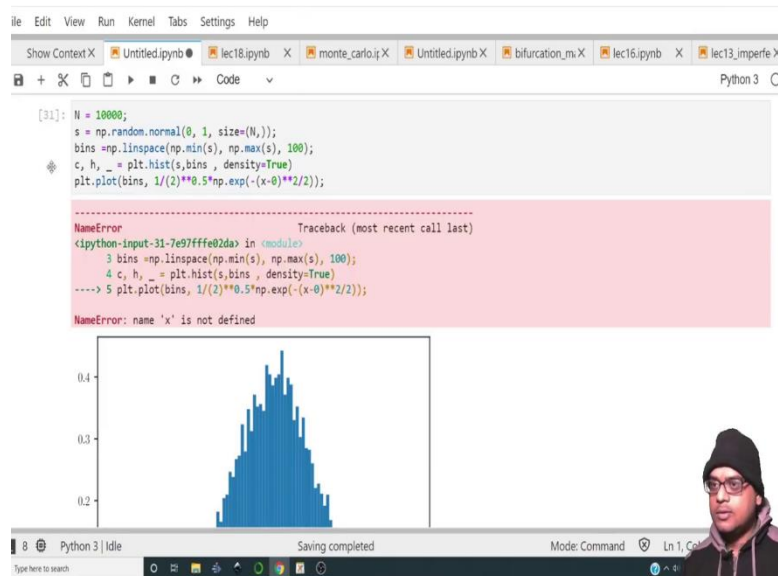
So, with none of this.

(Refer Slide Time: 17:58)



In fact, if I run this I can and again I will get a different histogram because each time I run this cell I am sort of calling this function again and again and I am getting a new distribution, alright. So, let us also plot on top of it the analytical expression for the probability density function.

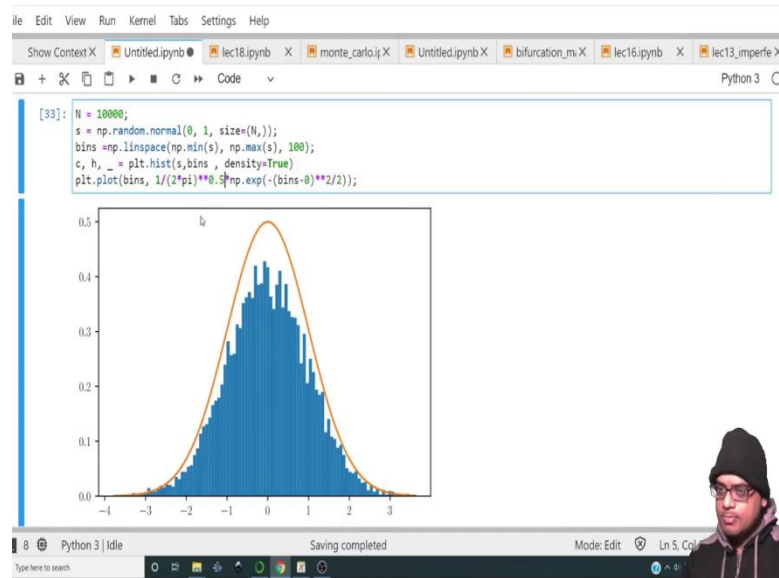
(Refer Slide Time: 18:20)



So, let me remove this and call this bins. Let me declare sorry bins equal to np dot let me copy the code. And, let me also plot on top of the on top of this plt.plot(bins, 1/sqrt(2*sigma**2) * np.exp(-(x-0)**2/2). So, this will because sigma is 1. So, this will be simply

2 to the power 0.5 times exponential $-(x - 0)**2$. So, x is not defined, this has to be bins
alright. `plt.plot(bins, 1/2**0.5*np.exp(-(bins-0)**2/2))`

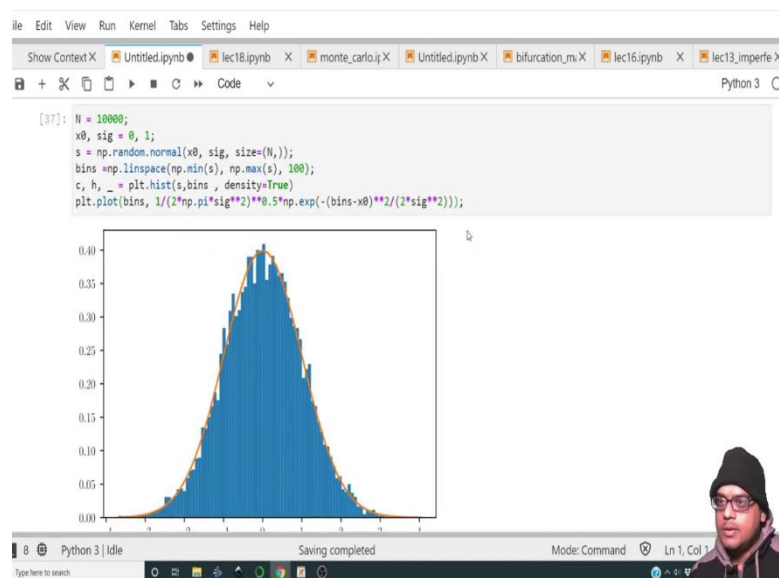
(Refer Slide Time: 19:22)



So, it seems I have obtained the magnitude incorrectly. I think there is a pi somewhere ok.

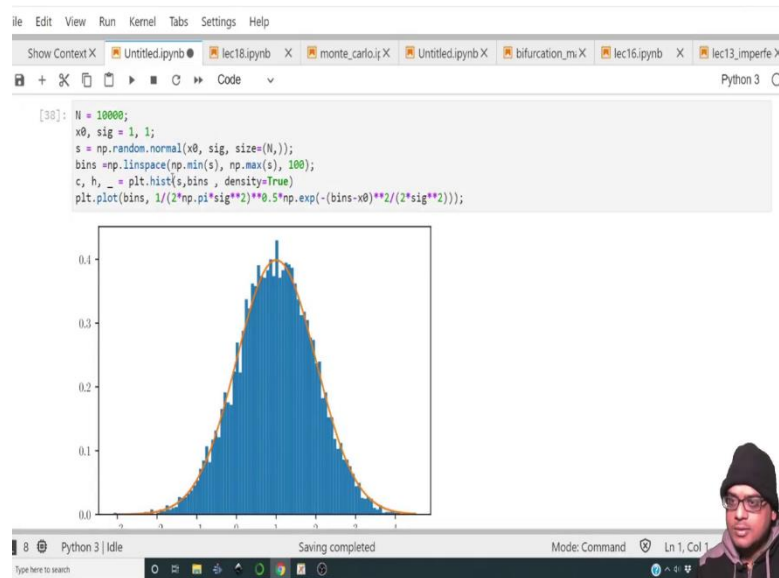
So, it is $1/(\sqrt{2\pi})e^{(-x^2/2)}$ and that gives us the appropriate function.

(Refer Slide Time: 19:57)



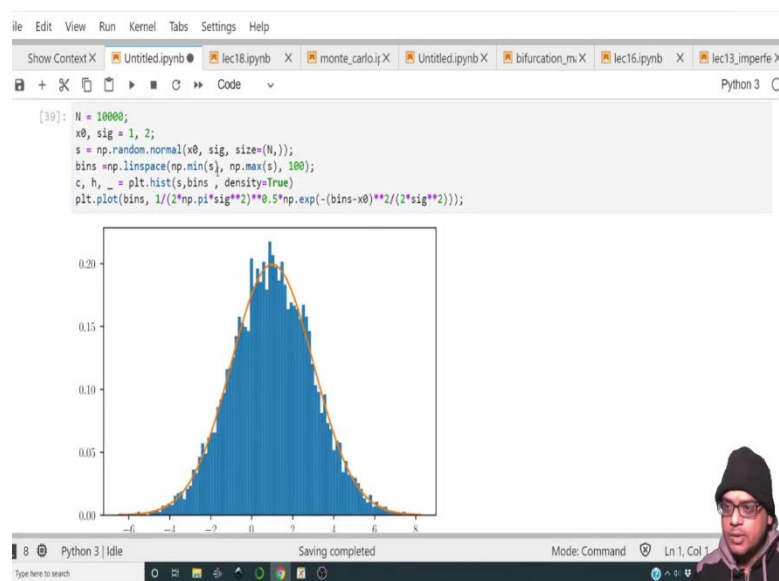
In fact, let me write σ square and let me write and let me define this as x_0 . So, now, we will give x_0 and σ and we will define $x_0, \sigma = 0, 1$. I think we have, yeah. So, now let me change the offset, let me make it 1.

(Refer Slide Time: 20:36)



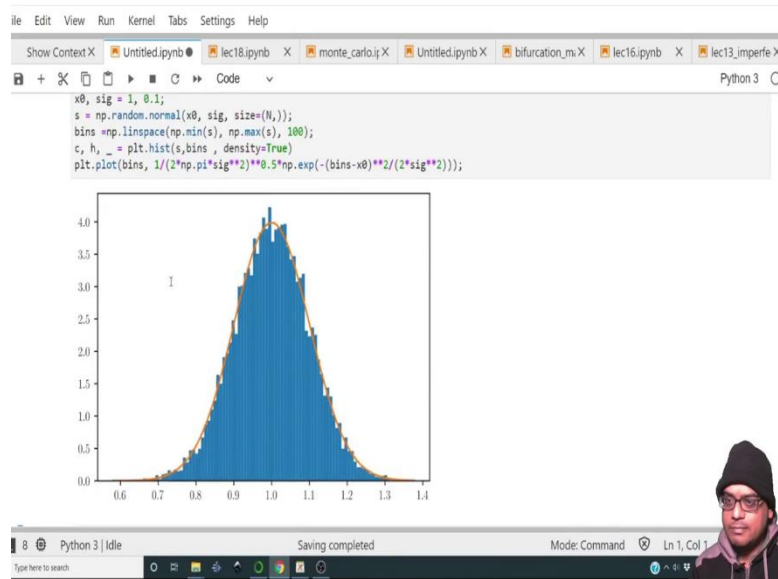
So, now, the Gaussian is centered around 1. So, you can have a look at the equation it is rather simple. It is I will write it down and you can have a look in the uploaded file. Well, this defines the offset of the Gaussian and this defines the width of the Gaussian.

(Refer Slide Time: 20:54)



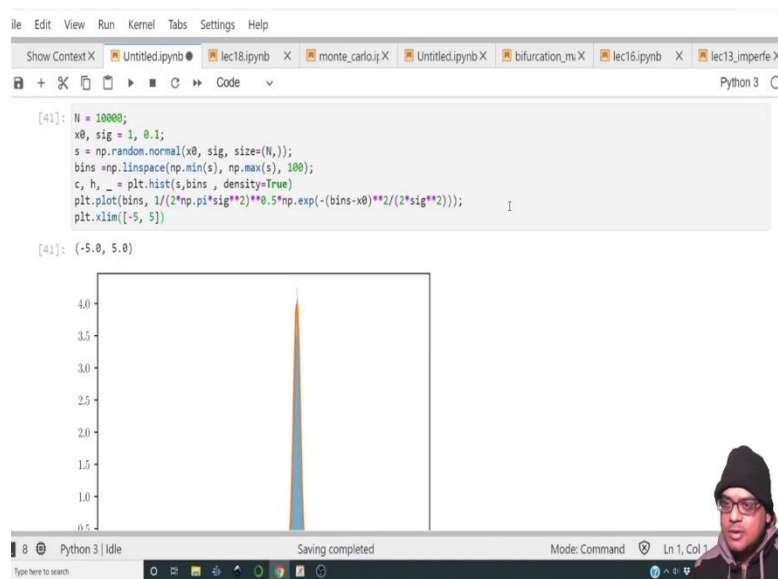
So, if I make it 2 so, then I have a larger spread of the Gaussian.

(Refer Slide Time: 21:03)

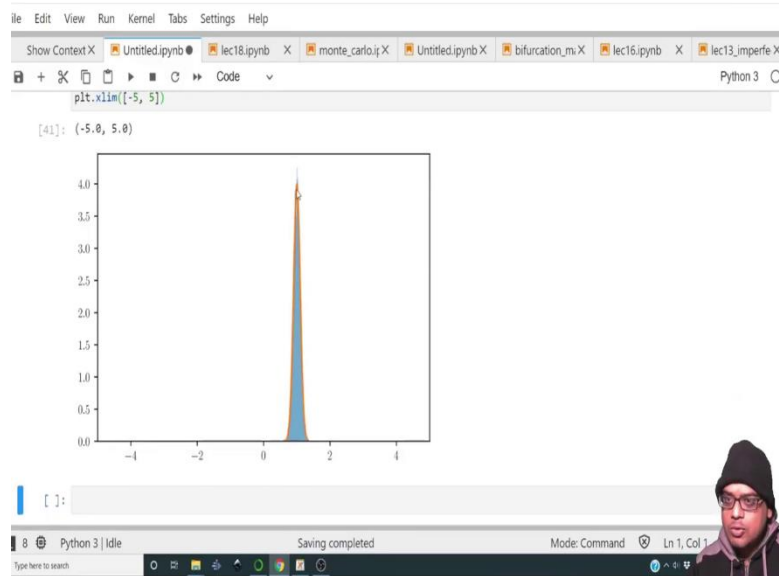


If I make it smaller I will have a lower spread of the Gaussian ok, it becomes much thinner.

(Refer Slide Time: 21:11)

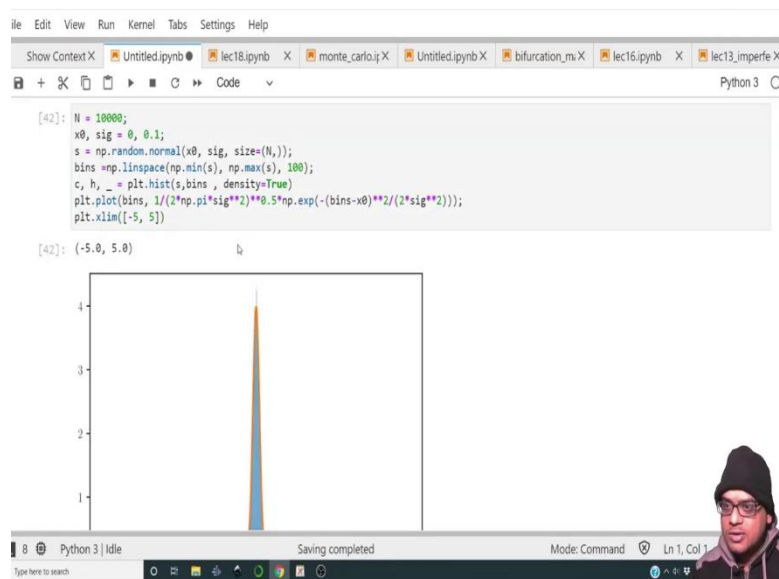


And, in order to really see whether it is becoming thinner or not I will just change I will limit the x axis. So, plt.xlim let it go from - 5 to 5. (Refer Slide Time: 21:20)



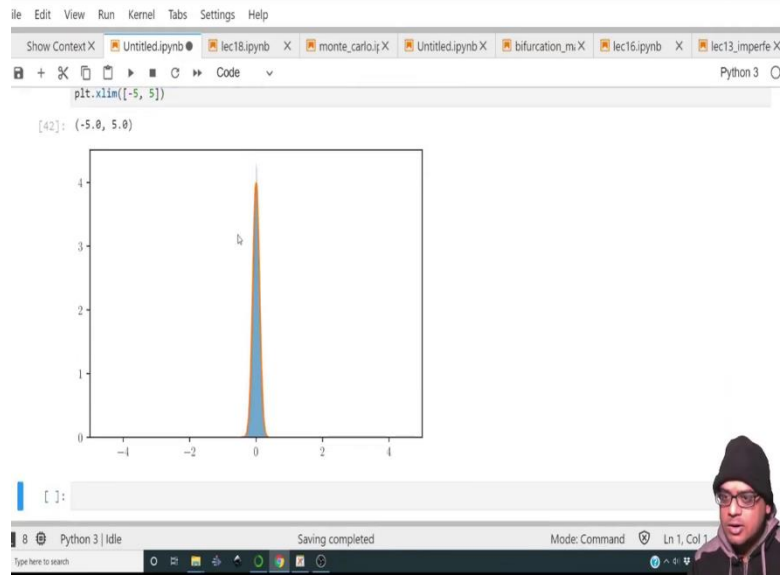
So, now it appears as a very thin Gaussian centered around 1.

(Refer Slide Time: 21:25)



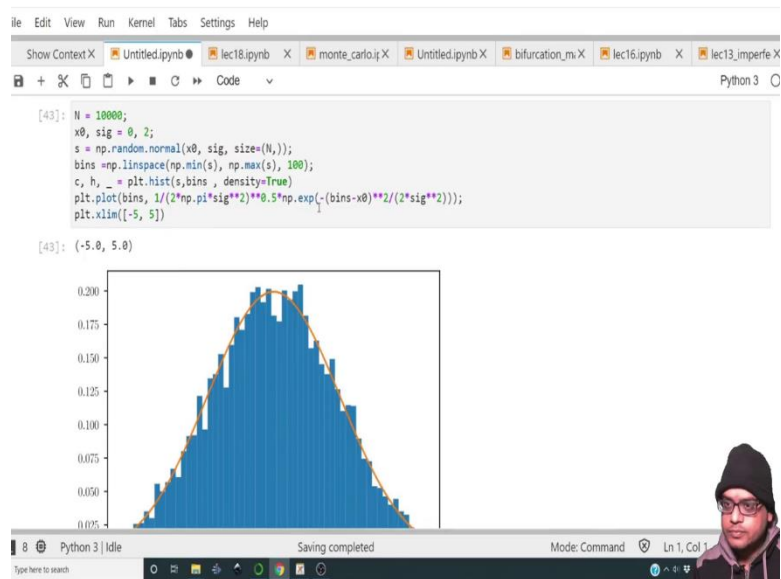
Let me make this 0 so that it is centered around the origin.

(Refer Slide Time: 21:27)

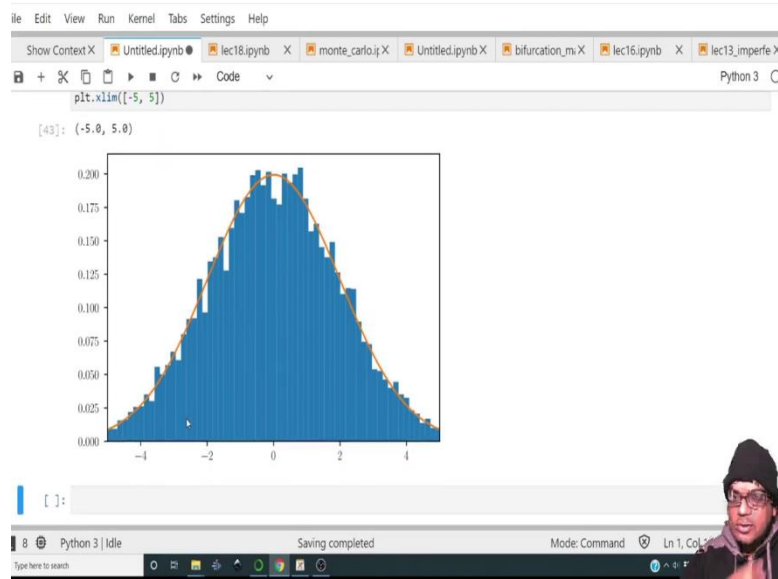


It was centered around yeah, it was centered around 1. Let me now increase this to 2.

(Refer Slide Time: 21:33)

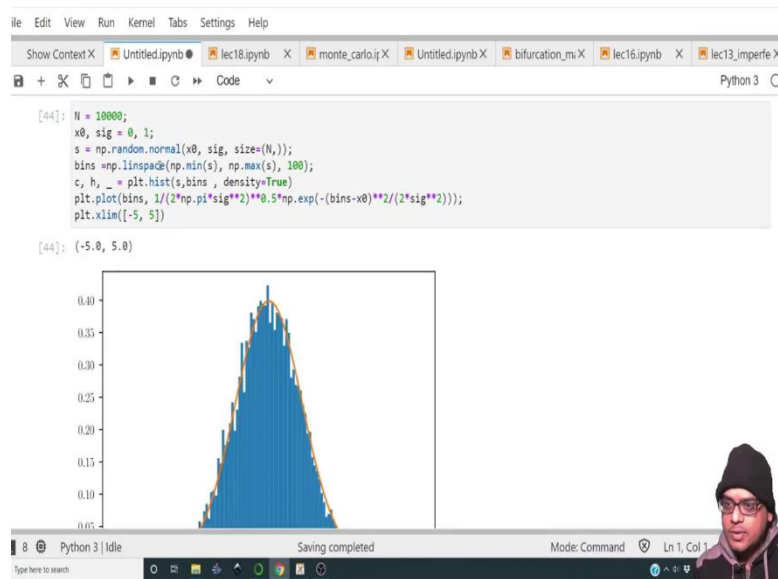


(Refer Slide Time: 21:35)



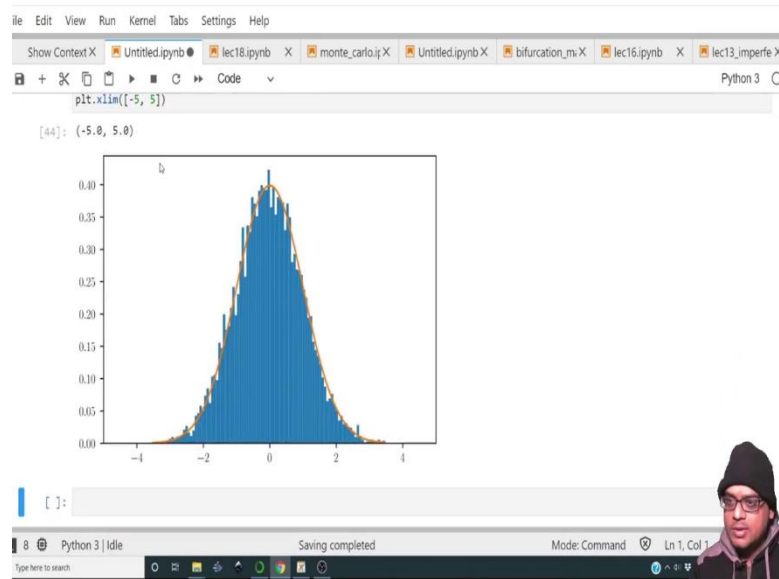
So, now, it has increased the x limits are still the same.

(Refer Slide Time: 21:41)



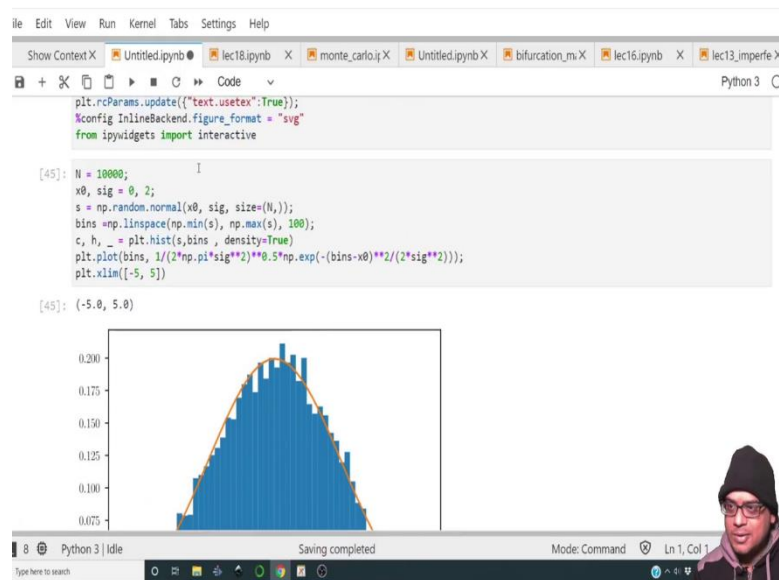
Let me make it 1.

(Refer Slide Time: 21:43)



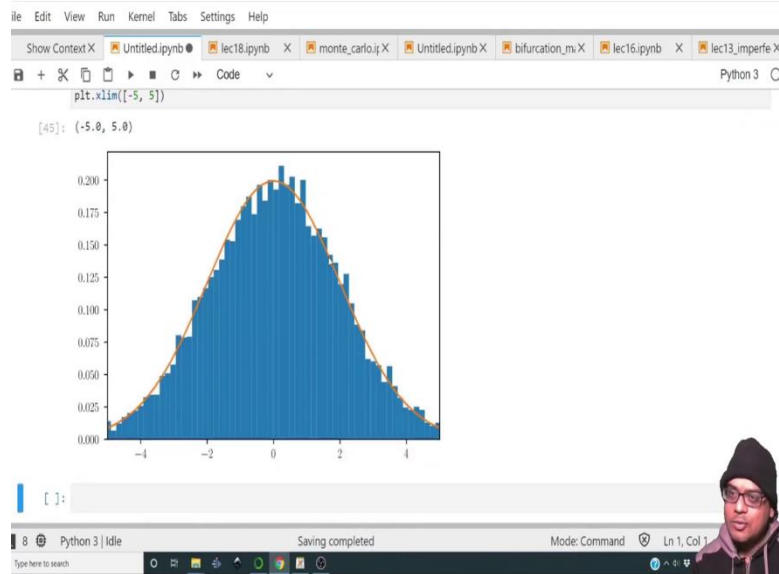
And, it is something like this. In fact, what we can do is we can plot a bunch of histograms like this for by looping over this. And, I will leave this as a small exercise to you. Just make a small loop in which you are changing sigma and seeing how the Gaussian changes.

(Refer Slide Time: 22:09)



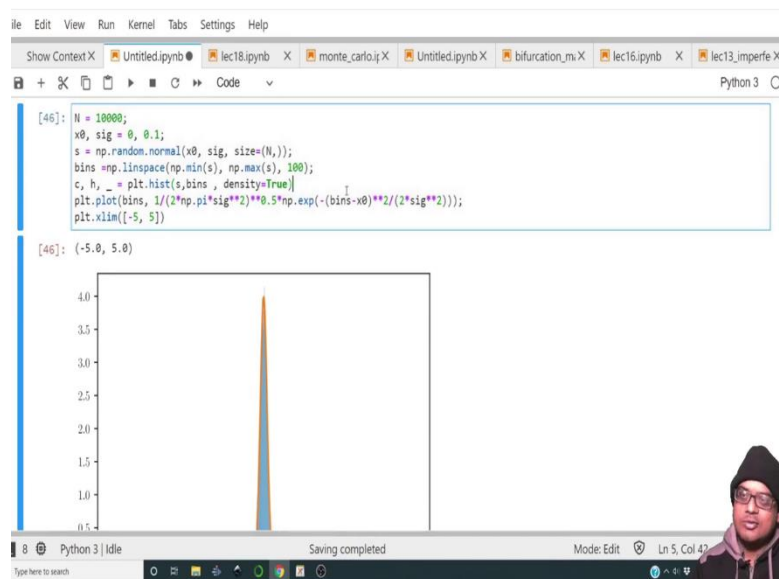
Notably once you make it fatter the maximum value will also change.

(Refer Slide Time: 22:09)



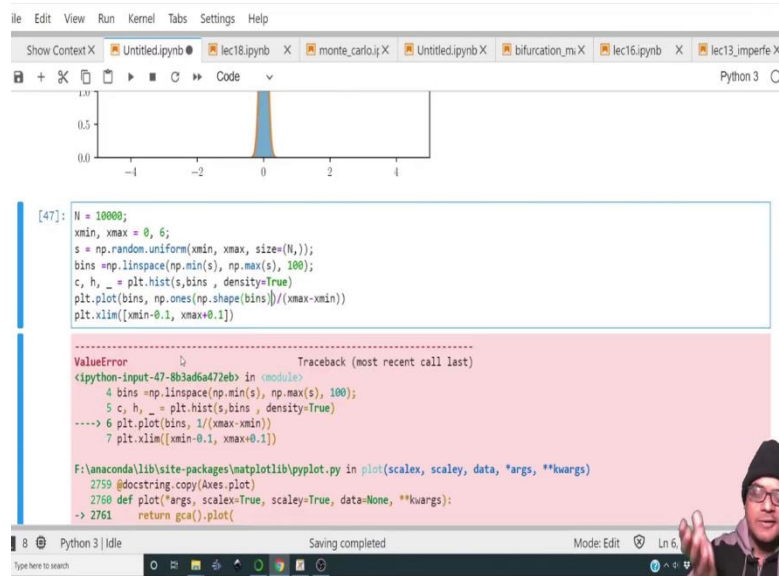
Can you guess why? Think about it.

(Refer Slide Time: 22:14)



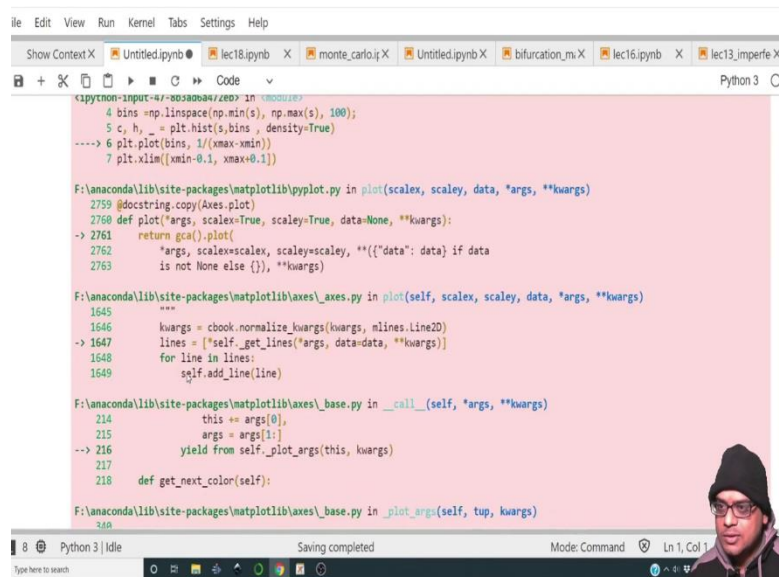
And once you make it narrower the maximum value will increase, it becomes 4. So, think about that. It is not that difficult to deduce. It has something to do with the fact that the integral of the pdf has to be 1 that the area under the curve has to be 1, right.

(Refer Slide Time: 22:36)

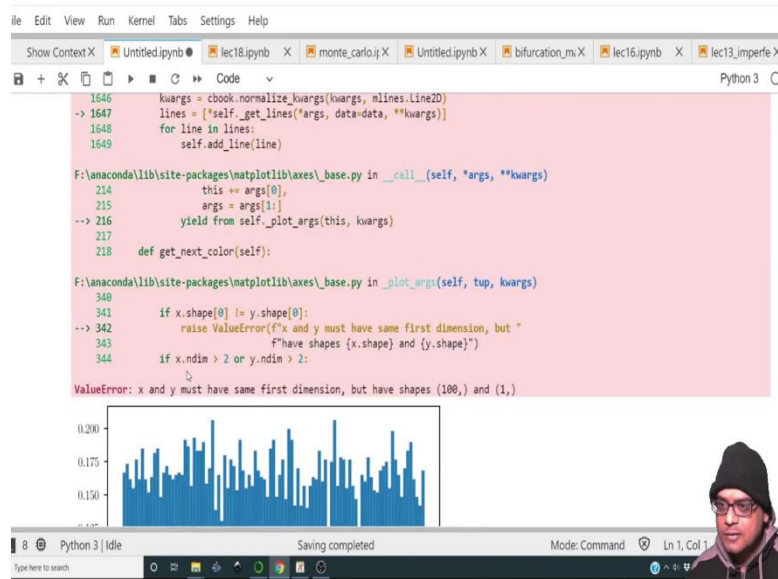


So, now if I go back to the uniform distribution, let me copy this and let me make this as xmin and make this xmax or let it go from 0 to 6. I will change this to xmin, xmax and we do not really want to plot this, but instead of this we will simply plot 1 upon xmax - xmin and the limit will be xmin - 0.1 to xmax + 0.1 and I am offsetting the limits of the plot because I want to show the bin edges as well. So, let me run this.

(Refer Slide Time: 23:31)



(Refer Slide Time: 23:32)



```
1646 kwargs = cbook.normalize_kwargs(kwargs, mlines.Line2D)
-> 1647 lines = ["self._get_lines(*args, data=data, **kwargs)"]
1648 for line in lines:
1649     self.add_line(line)

F:\anaconda\lib\site-packages\matplotlib\axes_base.py in __call__(self, *args, **kwargs)
214     this += args[0],
215     args = args[1:]
-> 216     yield from self._plot_args(this, kwargs)
217
218     def get_next_color(self):

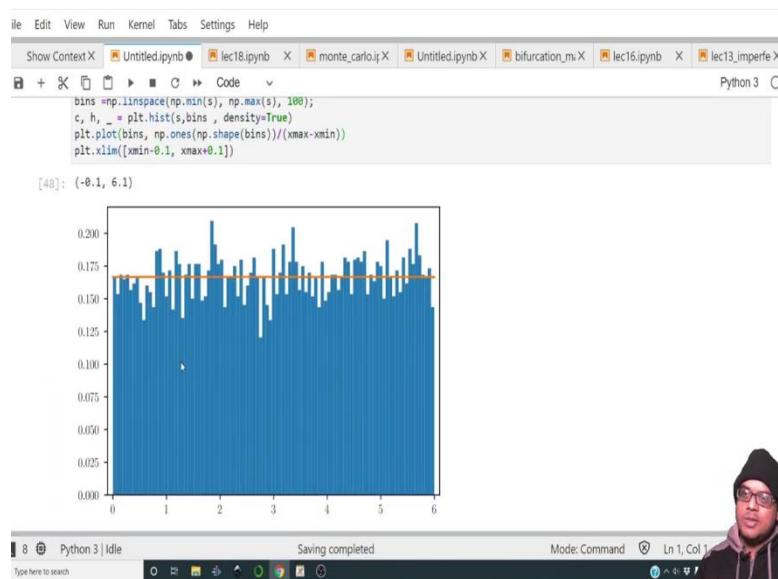
F:\anaconda\lib\site-packages\matplotlib\axes_base.py in _plot_args(self, tup, kwargs)
340
341     if x.shape[0] != y.shape[0]:
-> 342         raise ValueError(f"x and y must have same first dimension, but "
343             f"have shapes {x.shape} and {y.shape}")
344     if x.ndim > 2 or y.ndim > 2:

ValueError: x and y must have same first dimension, but have shapes (100,) and (1,)
```

The plot shows a histogram with blue bars. The y-axis ranges from 0.150 to 0.200. The x-axis ranges from 0 to 6. A small inset image of a person is visible in the bottom right corner of the notebook interface.

It seems we have a small error.

(Refer Slide Time: 23:40)



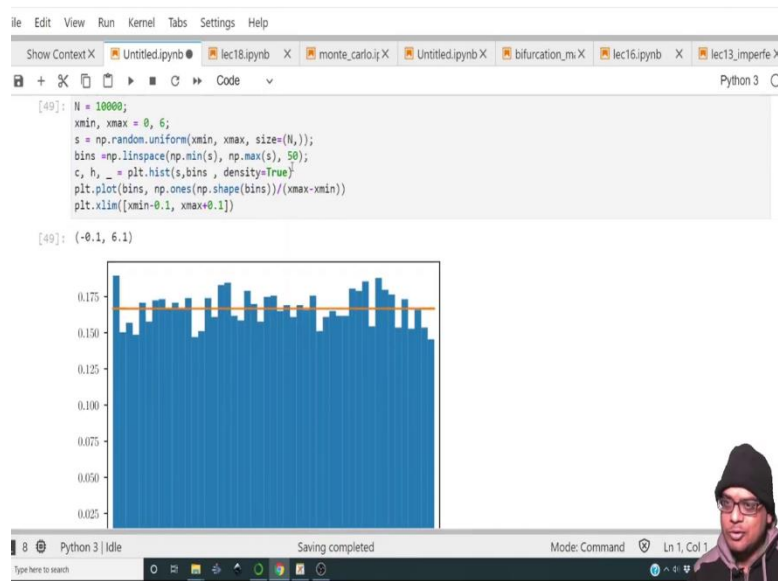
```
bins = np.linspace(np.min(s), np.max(s), 100);
c, h, _ = plt.hist(s, bins, density=True)
plt.plot(bins, np.ones(np.shape(bins))/(xmax-xmin))
plt.xlim([xmin-0.1, xmax+0.1])

[40]: (-0.1, 6.1)
```

The plot shows a histogram with blue bars. The y-axis ranges from 0.000 to 0.200. The x-axis ranges from 0 to 6. A horizontal orange line is drawn across the plot at approximately y = 0.167. A small inset image of a person is visible in the bottom right corner of the notebook interface.

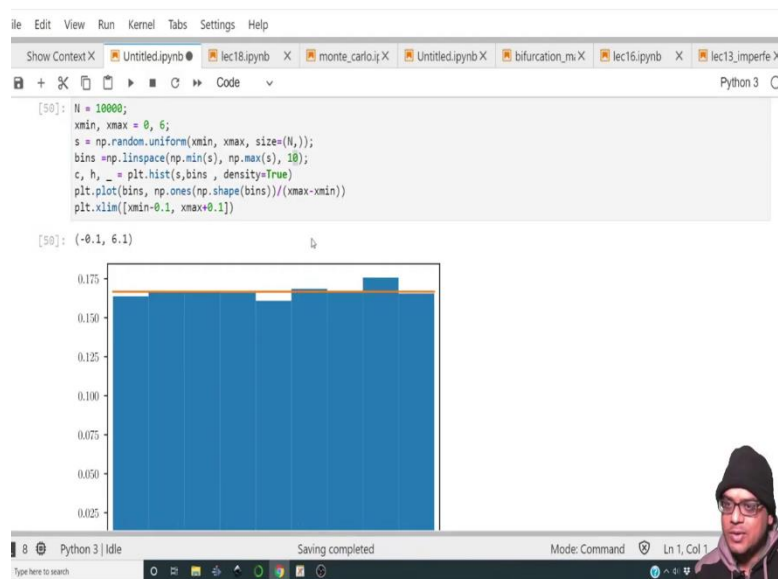
This has to be $\text{np.ones}(\text{np.shape}(\text{bins}))$. So, we cannot plot a vector against a scalar. So, what I had written initially was $1/(\text{xmax} - \text{xmin})$, but it has to be multiplied by a number of ones so that you can have two vectors which you can plot ok. So, this is a very classic mistake, but nothing to be worried about alright, great. So, this is now the uniformly distributed variables. Obviously, there will be some variables which are more sample, but more or less you have a uniform distribution.

(Refer Slide Time: 24:24)



If I increase or if I can reduce the number of bins it should become more uniform.

(Refer Slide Time: 24:29)



If I take only 10 bins then it becomes more and more uniform. So, think about it why reducing the number of bins makes it more uniform. So, these are the certain questions which if once you start playing around you will get an idea about it, alright. So, let us proceed further. Let us quickly take a look at the Pareto distribution because I want to establish a very useful concept by that.

(Refer Slide Time: 24:59)

Docstring:
pareto(a, size=None)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding 1 and multiplying by the scale parameter `m` (see Notes). The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is `mu`, where the standard Pareto distribution has location `mu = 1`. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

.. note::
New code should use the `pareto` method of a `default_rng()` instance instead; see `random-quick-start`.

Handwritten notes:
 - A number line from 1 to 10 with a box around 5.
 - A graph of the Pareto distribution curve $f(x)$ vs x , starting at $m=1$ and decaying towards zero. The curve is labeled with a and $(1 + \text{pareto})m$.
 - The formula: $f(x) = \frac{am^a}{x^{a+1}}$ for $x \in [m, \infty)$.

So, the Pareto distribution is one class of factorial distribution where the distribution looks something like this, right. So, in this distribution you must supply the scale of this distribution that is m and the power law that it follows that is a . So, the distribution, so, if this, the random variable is x and the distribution $f(x)$.

So, $f(x) = am^a/x^{a+1}$ will be a m to the power let me just yeah a divided by x raised to $a + 1$. So, this is the Pareto II distribution over m to infinity. So, the random variable can lie as between m and infinity; this can be open say alright. So, now, let us sample from the Pareto distribution and let us draw the histogram. It is quite simple, but regardless. Later on we will see manually how to do it.

(Refer Slide Time: 26:08)

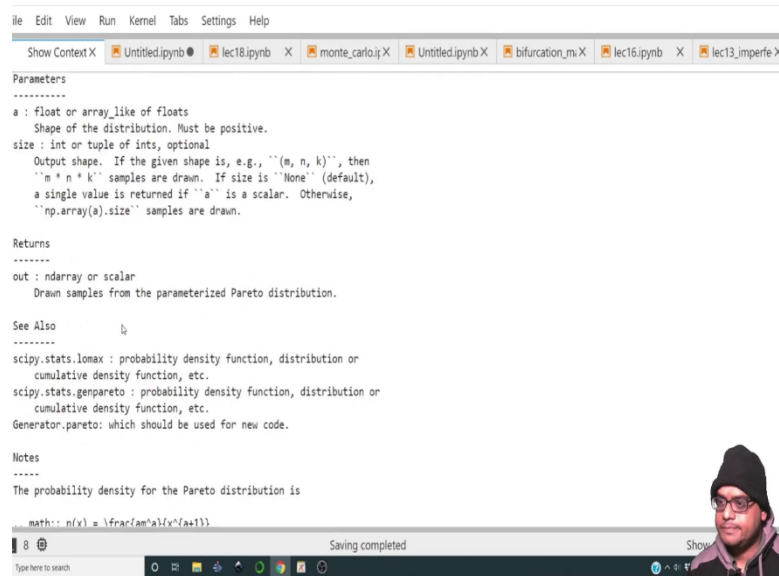
```

N = 10000;
xmin, xmax = 0, 6;
s = (1+np.random.pareto(xmin, xmax, size=(N,)))**m;
bins = np.linspace(np.min(s), np.max(s), 10);
c, h, _ = plt.hist(s, bins, density=True);
plt.plot(bins, np.ones(np.shape(bins))/(xmax-xmin));
plt.xlim([xmin-0.1, xmax+0.1])

```

So, this will simply become the pareto and let us quickly look at the contextual help for the pareto distribution.

(Refer Slide Time: 26:25)



```
file Edit View Run Kernel Tabs Settings Help
Show Context X Untitled.ipynb X lec18.ipynb X monte_carlo.ipynb X Untitled.ipynb X bifurcation_m X lec16.ipynb X lec13_imperfe X

Parameters
-----
a : float or array_like of floats
    Shape of the distribution. Must be positive.
size : int or tuple of ints, optional
    Output shape. If the given shape is, e.g., ``(8, n, k)``, then
    ``m * n * k`` samples are drawn. If size is ``None`` (default),
    a single value is returned if ``a`` is a scalar. Otherwise,
    ``np.array(a).size`` samples are drawn.

Returns
-----
out : ndarray or scalar
    Drawn samples from the parameterized Pareto distribution.

See Also
-----
scipy.stats.lomax : probability density function, distribution or
cumulative density function, etc.
scipy.stats.gempareto : probability density function, distribution or
cumulative density function, etc.
Generator.pareto : which should be used for new code.

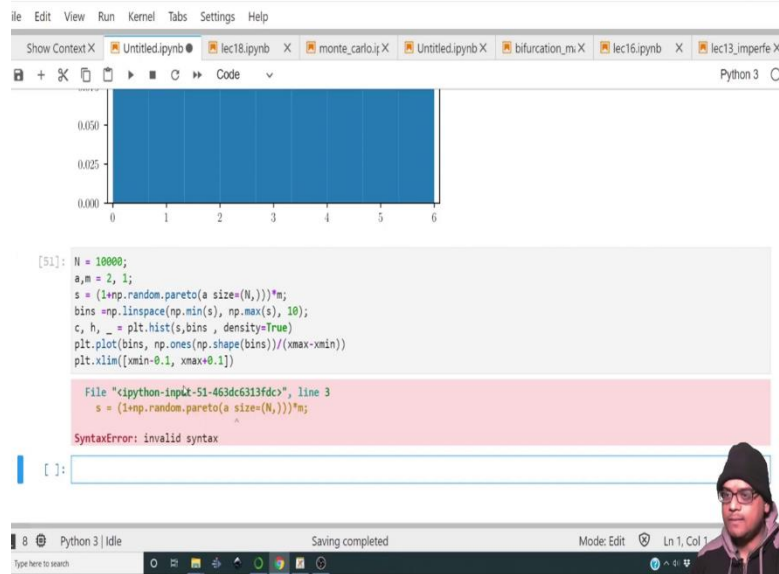
Notes
-----
The probability density for the Pareto distribution is

math:: n(x) = \frac{m^a}{\Gamma(a)} x^{-a-1}
```

So, it is simply taking a and the size. So, because it is only taking the power law index what we can do is we can do this and by default it generates the distribution from 1, ok. So, the mu what this peak over here is equal to 1 and in fact, the generalized Pareto distribution is available inside SciPy, but let us use the NumPy distribution for now.

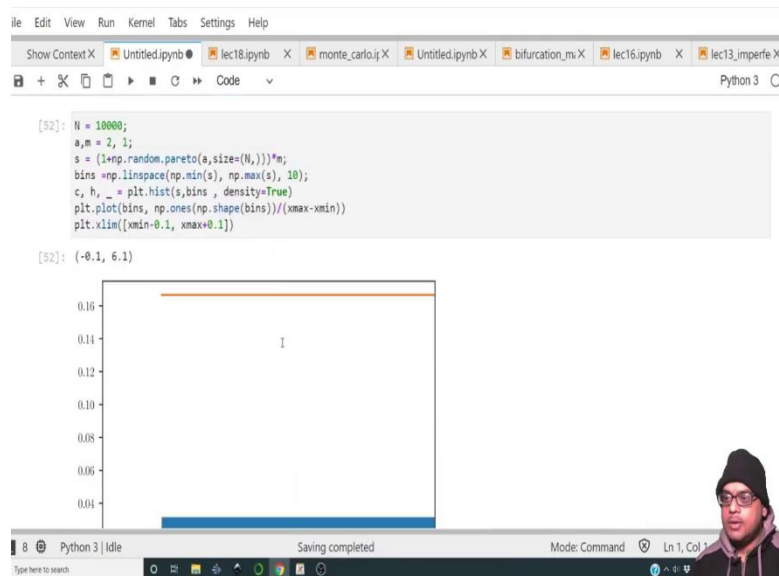
So, what we will simply do is because we are generating something with 1, or in fact, ok. So, the classical Pareto distribution can be obtained by adding 1 and multiplying the scale factor. So, we will do 1 + pareto and multiply it with our factor m alright, fair enough.

(Refer Slide Time: 27:31)



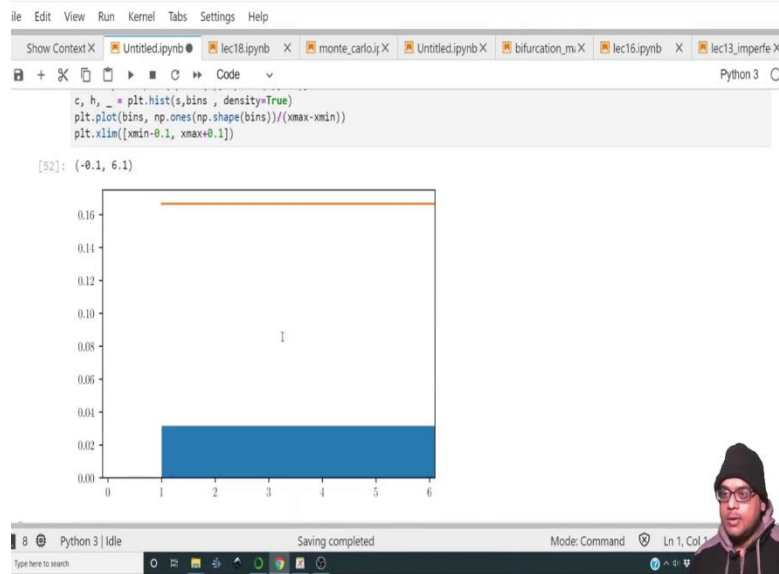
So, 1 plus this multiplying m. So, this will give us the appropriate Pareto distribution and the only index that we need is going to be a. So, a comma m is going to be say 0 or 2 comma 1, alright. So, let me run this and see.

(Refer Slide Time: 28:01)

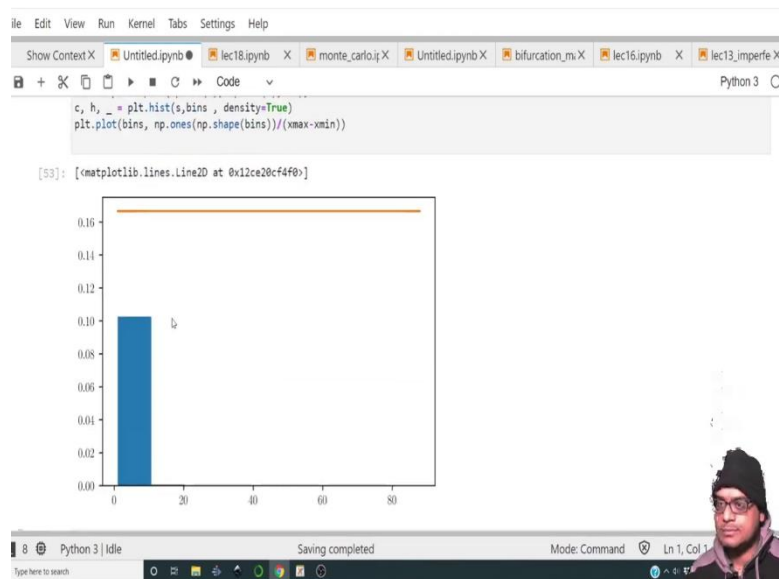


Have we missed bracket somewhere; we missed a comma, alright.

(Refer Slide Time: 28:03)

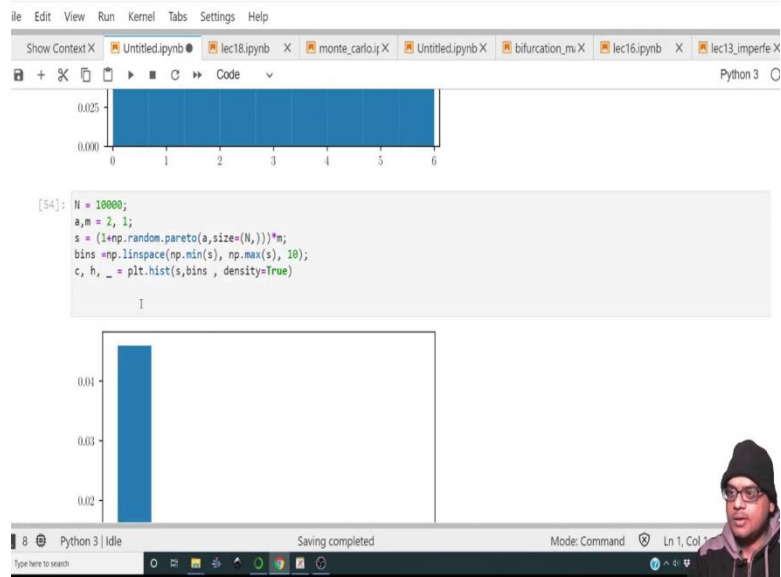


(Refer Slide Time: 28:11)

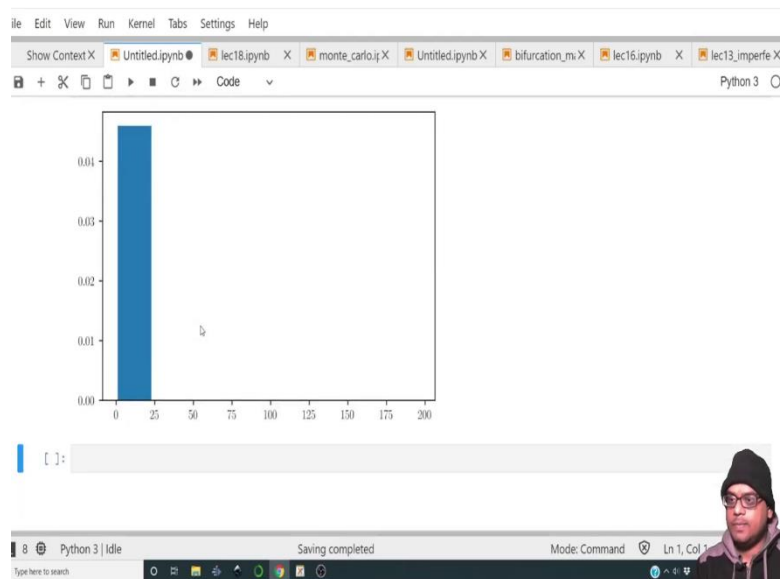


So, there is no. We do not need the uniform distributions as well.

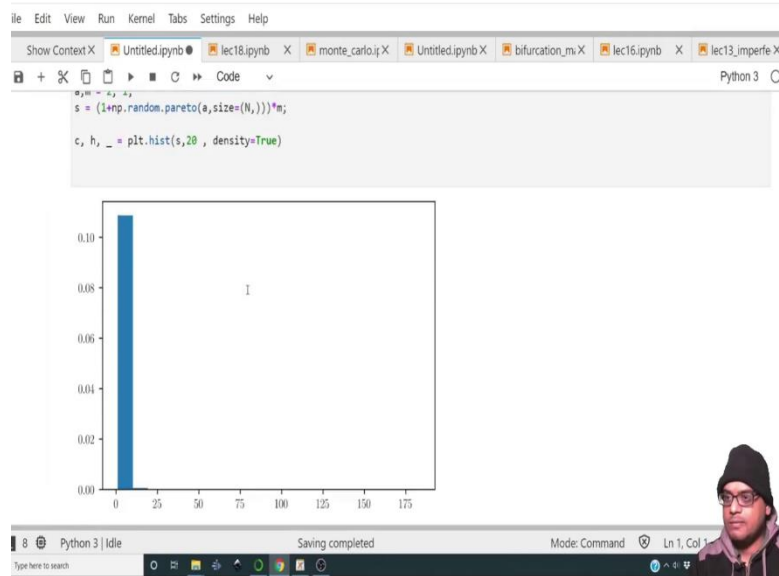
(Refer Slide Time: 28:19)



(Refer Slide Time: 28:19)

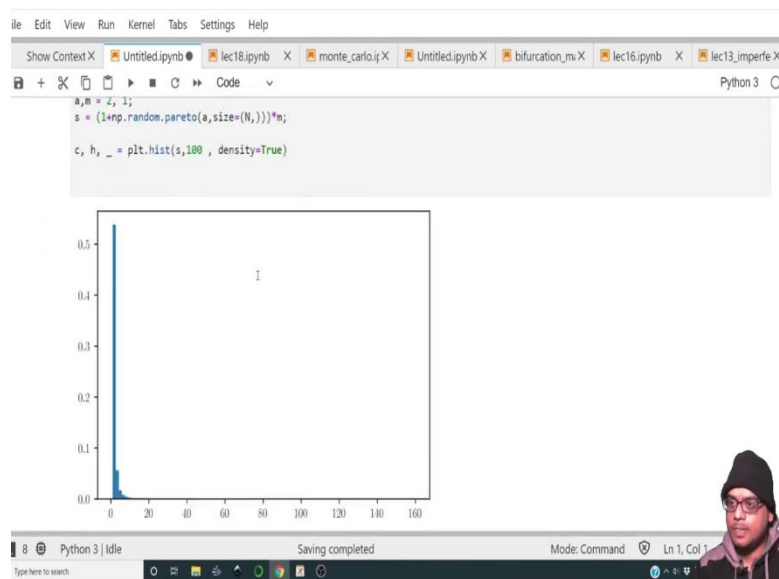


(Refer Slide Time: 28:30)



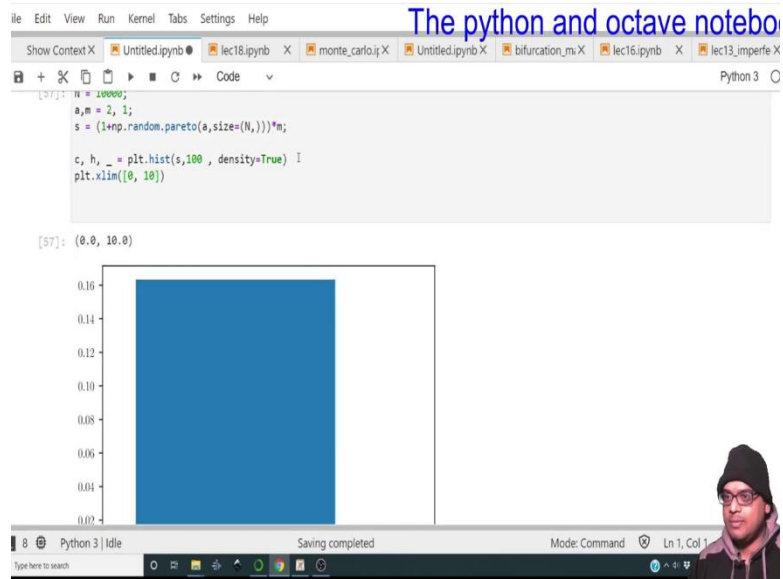
And let us remove the bins and let me take 20 bins.

(Refer Slide Time: 28:34)



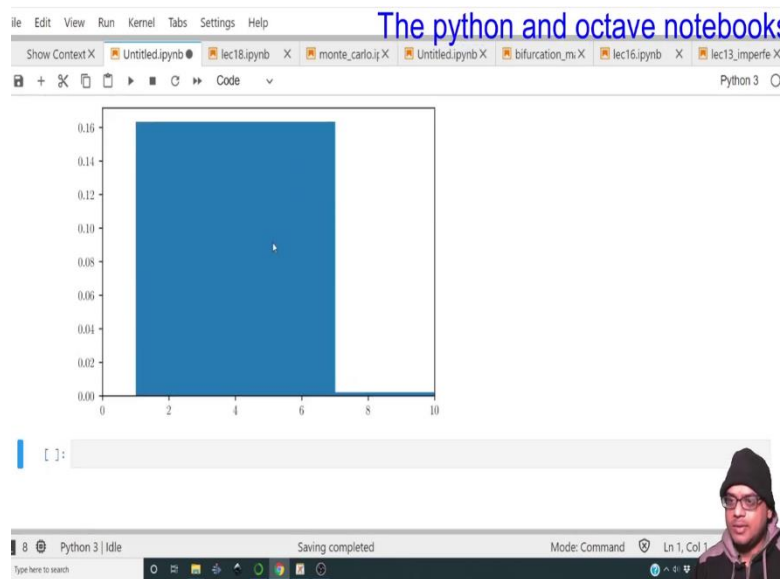
Now, let me take 100 bins alright. So, there is a very high probability of something near 1 and then quickly decays to something like this. And, it has nonzero probability for large values and this is what we will see over here the probability is decaying, but even for large values it is small, but it is finite. It is not going to 0 that is why you have this.

(Refer Slide Time: 29:06)

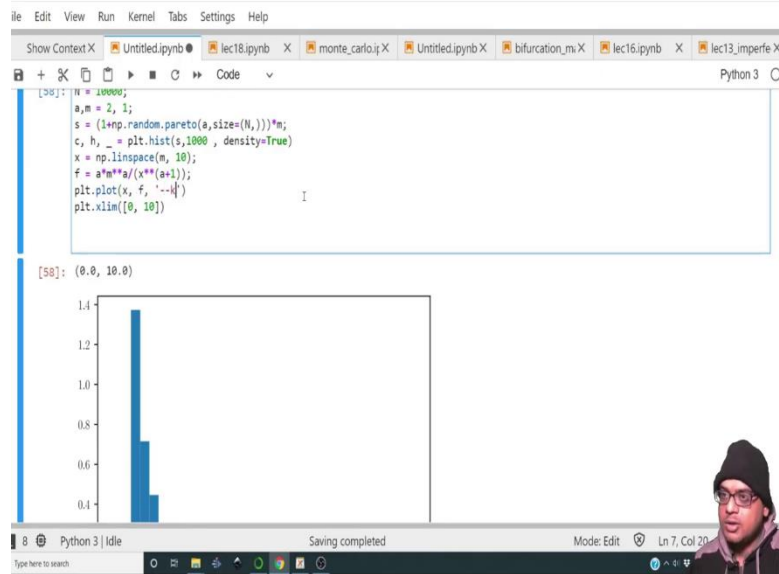


Let us limit the x-axis scale to go from 0 to 10 maybe.

(Refer Slide Time: 29:12)

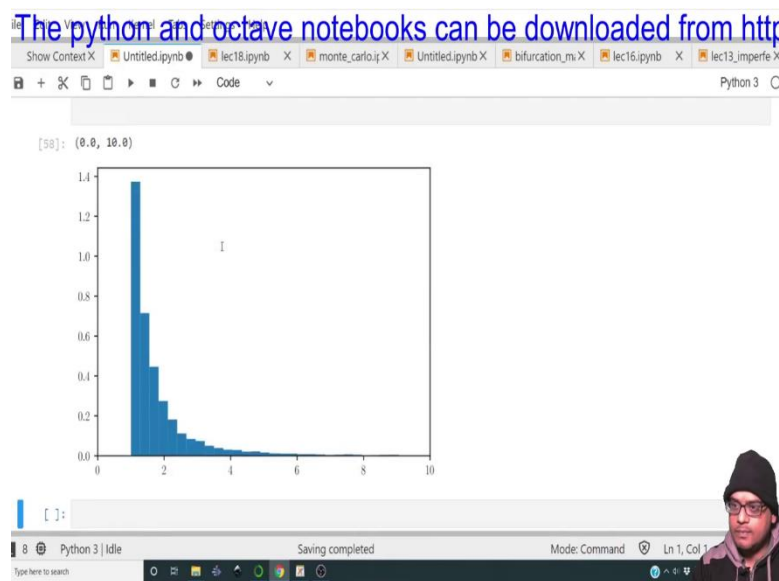


(Refer Slide Time: 29:20)



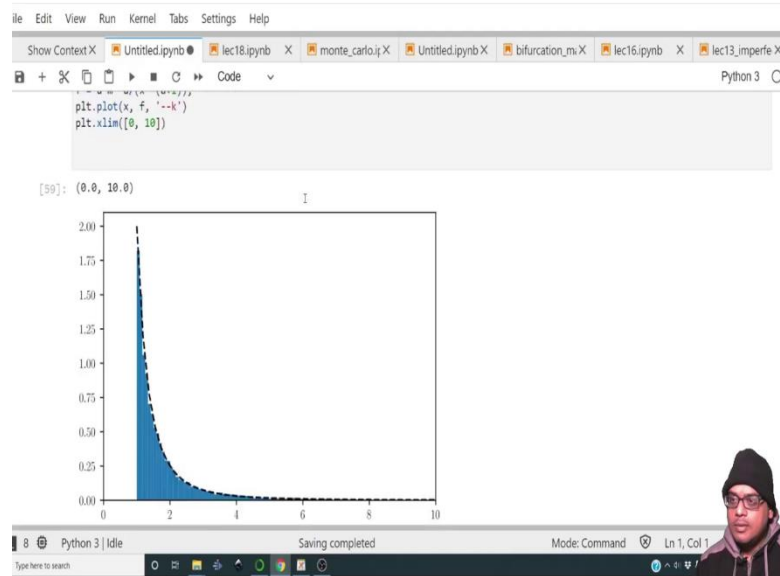
And let us take a 1000 bins ok.

(Refer Slide Time: 29:21)



So, it looks something like this. So, now, let us plot the distribution on top of it. So, what do we have? Something between 0 and 10. So, let me define x as np.linspace 0 to 10 then my the pdf f will be a times m raised to a divided by x to the power a + 1, alright. So, this has to be from 1, actually it has to be from m. So, let me run this and in fact, let me we have to plot this as well. So, plt dot plot x comma f alright.

(Refer Slide Time: 30:11)

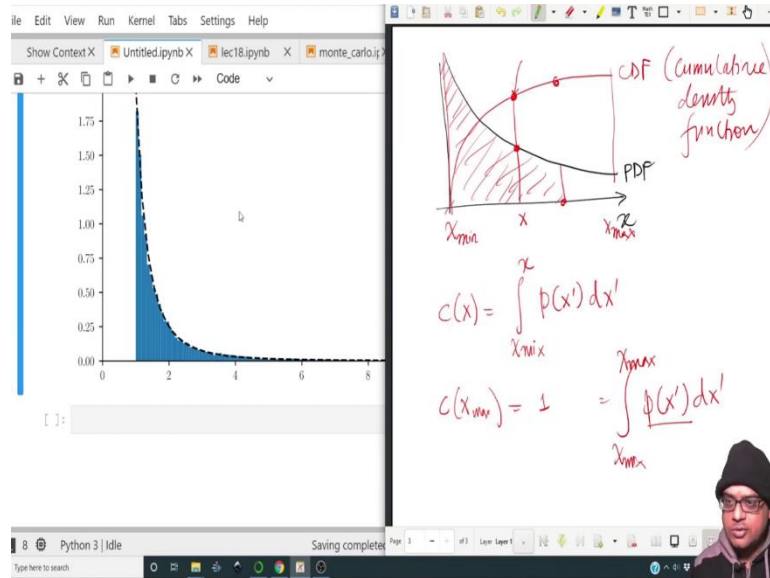


Let me make it a broken black line great. So, the analytical expression that we have written over here this particular expression and a large number of samples that we plucked from the Pareto distribution of NumPy they are in good agreement and it should not come as a surprise because in then we are sampling from the distribution and then eventually plotting it.

So, it should not come as a great surprise to you, but I am just showing you some of the nitty gritty of how that a , m all these things are decided. So, each distribution has its own sort of set of parameters that you need to pass along and depending on the work you have to supply the appropriate parameter in this case it was a and m .

And, the default function that NumPy gives it is choosing m has to be 0 we only give a , but we can always do this little trick and see in the help itself tells you to do that. So, but you can use the generalized Pareto distribution from SciPy if you do not want to do this little thing alright. So, now, I am going to show you how to sample from a random distribution or some probability distribution function. So, the basic idea is as follows.

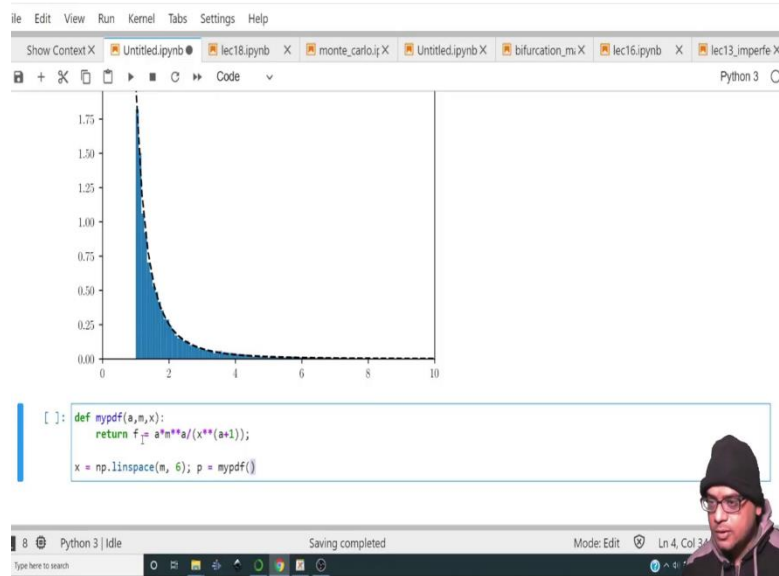
(Refer Slide Time: 31:44)



So, if you have a PDF. So, if this is x , suppose this is the PDF. So, this is the PDF. So, what we will do is we will plot this PDF. So, this is called as the CDF. So, if this is X_{min} and this is X_{max} . So, the CDF so, $C(x)$ is defined as integral X_{min} to x of $p(x')$ dx' . $C(x) = \int_{x_{min}}^x p(x') dx'$. So, if this is the value x , so, the CDF this particular value will be the integration of the PDF from the minimum value to that particular value and this is assigned to this value.

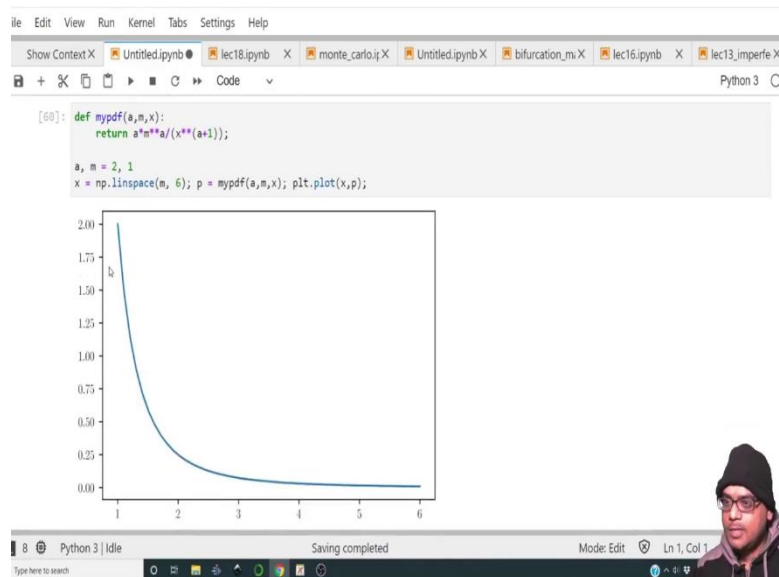
So, at this particular point it will be this particular integration and it will be assigned to this. So, obviously, when x becomes equal to X_{max} so, cumulate the C the value of the cumulative distribution function or the cumulative density function at X_{max} will be equal to 1, because it is essentially going to be integral of X_{min} to X_{max} of $p(x')$ dx' and by definition of being a probability density function, this integral is going to be 1, alright. So, let us take this distribution and try to work with it.

(Refer Slide Time: 33:28)



So, I am going to take this and in fact, let me wrap this inside a function. So, mypdf and let it return this and the inputs to mypdf will be a, m and x, right. So, now, let x be np.linspace it will go from m to set 6, we just want a truncated set of variables, alright. So, then f or rather the pdf p will be mypdf.

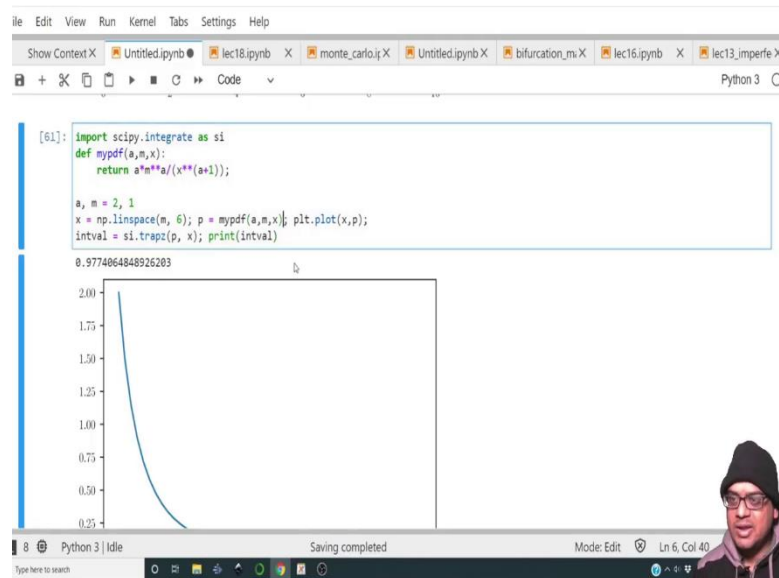
(Refer Slide Time: 34:15)



This has to be only this. So, this there is a certain risk you run when you are trying to reuse code, but anyway one has to be careful a, m, x, alright. So, let us define a ,m =2 ,1 alright. So, now, let us plot this plt.plot(x, p), alright.

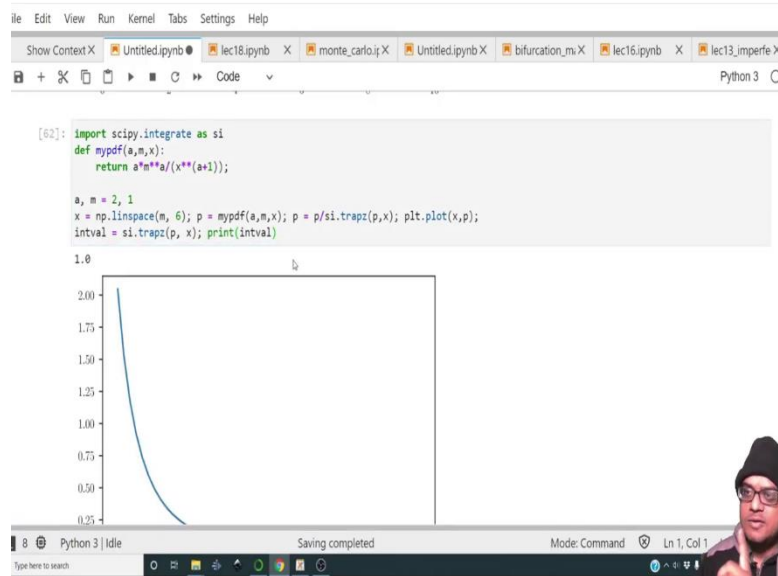
So, this is the distribution that we have, alright. So, now, it is only bounded to 1 to 6, but is it really going to be the pdf and the answer is no because let us quickly print out the integral of this supposed pdf from x min to x max.

(Refer Slide Time: 34:59)



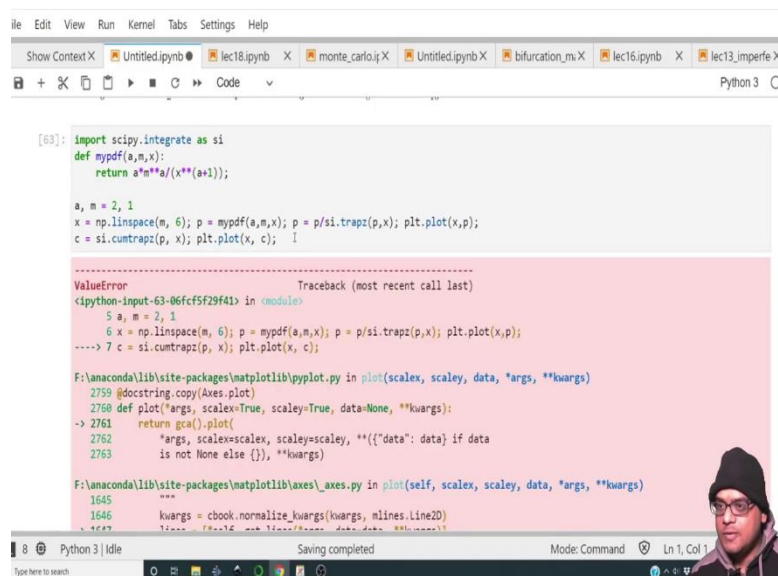
So, `intval` is equal to. So, for this for performing the integration we have to import `scipy.integrate` as `si` alright. So, `si.trapez` then this will be `si.trapez(p, x)`. So, then we will print `intval` alright. So, `intval` comes out to be 0.977. So, obviously, the value of the integral is not 1. So, we must what we call as normalized the pdf. So, we must divide it by the integral.

(Refer Slide Time: 35:39)



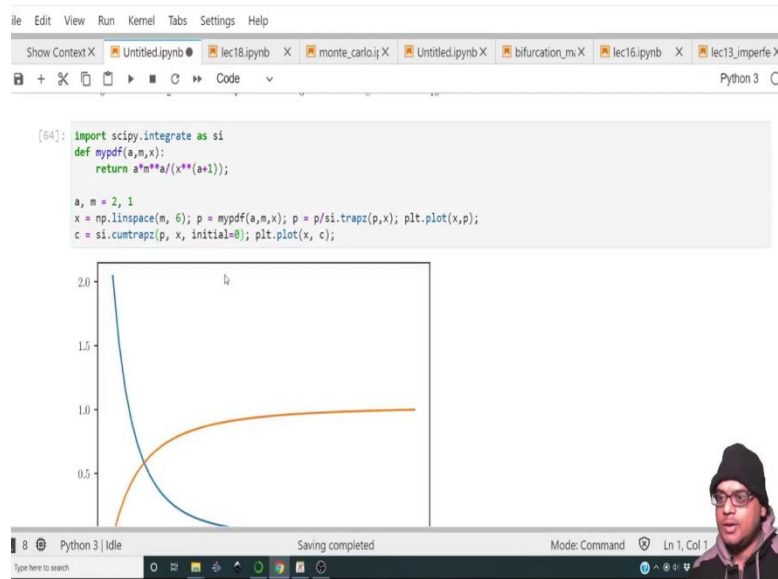
So, si.trapz or we can do the following then p equal to p divided by si.trapz(p,x). So, now, when we do this we have the value of the integral to be exactly 1. So, now, this function which spans over m to 6, where m is equal to 1 in this case that has a value equal to 1 that integral is equal to 1. So, it is a valid probability density function. If it does not have the integral equal to 1, it is not a valid probability density function, alright. So, we can get rid of this.

(Refer Slide Time: 36:28)



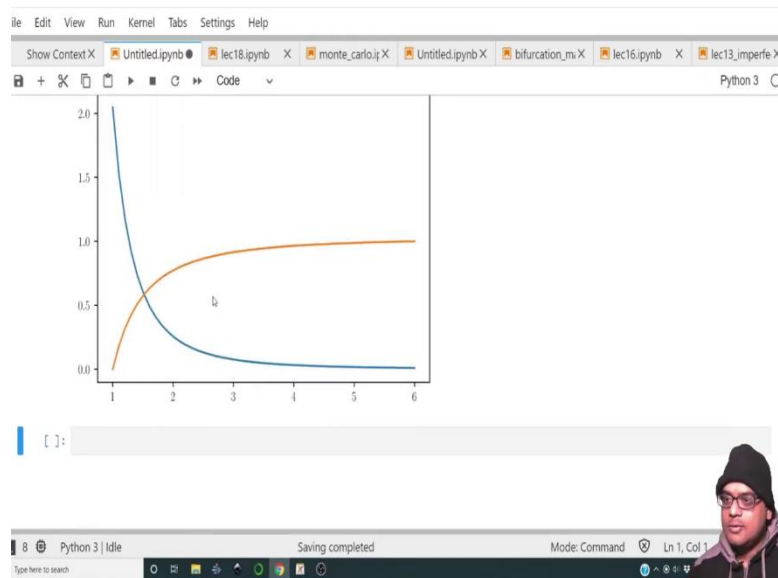
Now, let us plot the cumulative distribution function. So, c will be equal to si.cumtrapz and over here will be p comma x, then we will do a plot plt.plot(x, c) alright. Let us see what we get. We have an error.

(Refer Slide Time: 36:45)



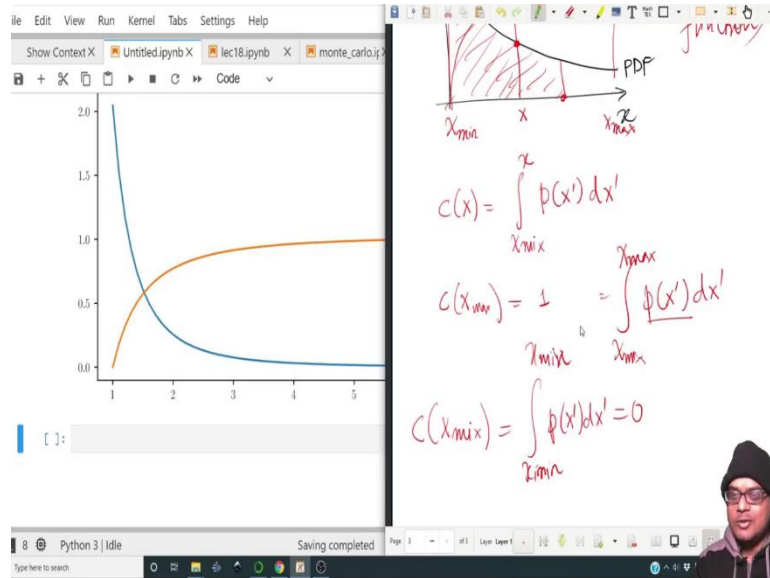
So, once we do a cumtrapz it ignores the initial point. So, we will write initial equal to 0, alright.

(Refer Slide Time: 37:00)



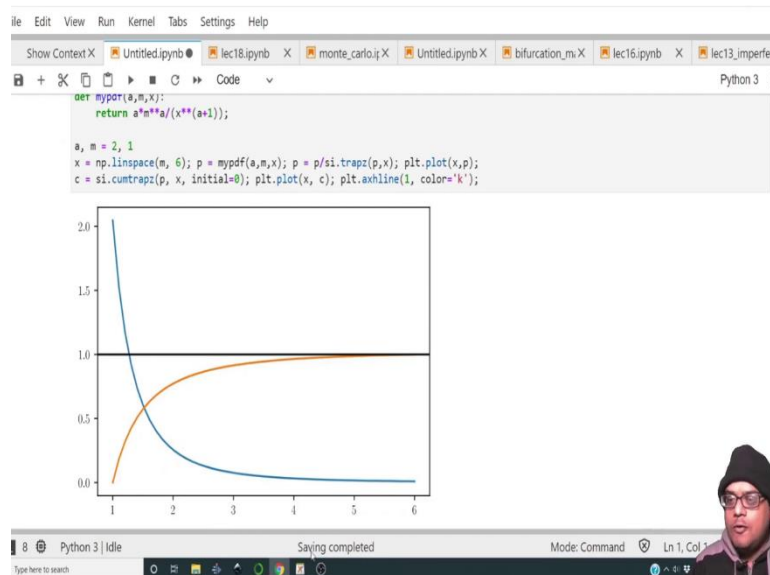
So, it ignores that initial point because the initial point is quite trivial.

(Refer Slide Time: 37:08)



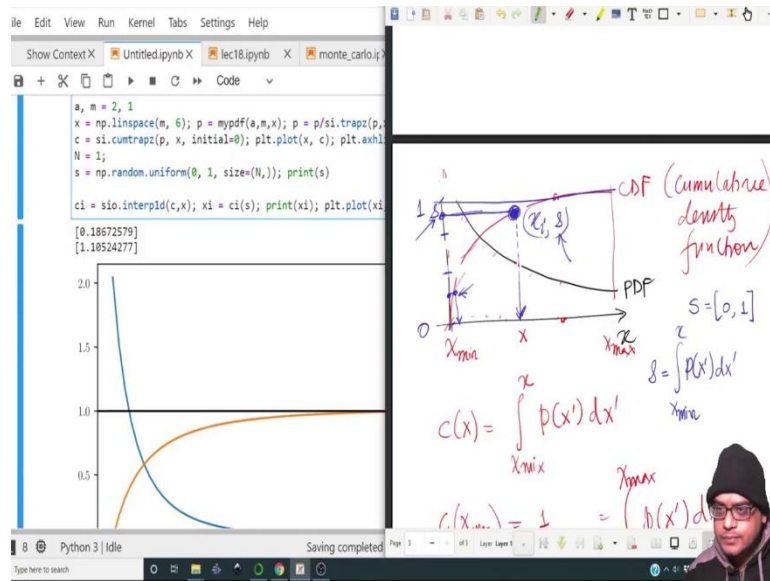
So, C of x is this. So, if I write $C(X_{min}) = \int_{x_{min}}^{x_{min}} p(x') dx'$. Now, obviously, this is 0 because the line itself has the integration range itself is 0. So, we have to declare that once you do that you put an initial value of 0 and this is something which you need to do. In octave I do not think you need to do this, alright.

(Refer Slide Time: 37:49)



So, we have this and in fact, let me also draw the line of 1. So, then we will do `plt.axvline` or the `hline 1`, alright and let me make it as a broken let me write `color equal to 'k'`, alright. So, that is that limit of 1. So, now, our task is I mean I have used a known pdf, but suppose that function were to be not known and I want to sample a lot of values from that function.

(Refer Slide Time: 38:24)



So, we have over here the let me go back to that figure, alright. So, let me erase all this. So, we have the PDF and we have the CDF. So, now, how do I sample points from this; that is the question. So, the way to do it is you know that the CDF will go all the way from 0 to 1. So, now, your task is to find out a random number uniformly distributed between 0 and 1.

So, essentially suppose this is that random number or this is that random number. So, now we drop this value on the cdf. Now, we see what is the corresponding X you have to that value. Essentially what we are saying is $s = \int_{x_{min}}^x p(x') dx'$. So, for what X do I obtain that value of this sampling or that random value a uniformly drawn random value? So, let us do that. Let me draw one uniformly one uniform; let me draw a uniformly sampled random number between 0 and 1.

(Refer Slide Time: 40:00)

The screenshot shows a Jupyter Notebook with the following code:

```

def mypdf(a,m,x):
    return a*m**a/(x**(a+1));

a, m = 2, 1
x = np.linspace(m, 6); p = mypdf(a,m,x); p = p/si.trapz(p,
c = si.cumtrapz(p, x, initial=0); plt.plot(x, c); plt.axh1
N = 1;
s = np.random.uniform(0, 1, size=(N,)); print(s)

```

The output of the code is `[0.16418196]`. Below the code is a plot showing the probability density function (PDF) and its cumulative distribution function (CDF). The PDF is a blue curve that starts at $x=1$ and decreases towards zero. The CDF is an orange curve that starts at $(1,0)$ and increases towards $(6,1)$. A horizontal line is drawn at $y=0.16418196$ on the CDF plot, and a vertical line is drawn from that point down to the x-axis, indicating the corresponding value of x .

Handwritten notes on the right side of the notebook explain the relationship between the CDF and the PDF:

$$c(x) = \int_{x_{min}}^{x_{mix}} p(x') dx'$$

$$c(x_{min}) = 0 = \int_{x_{min}}^{x_{min}} p(x') dx'$$

$$c(x_{max}) = 1 = \int_{x_{min}}^{x_{max}} p(x') dx'$$

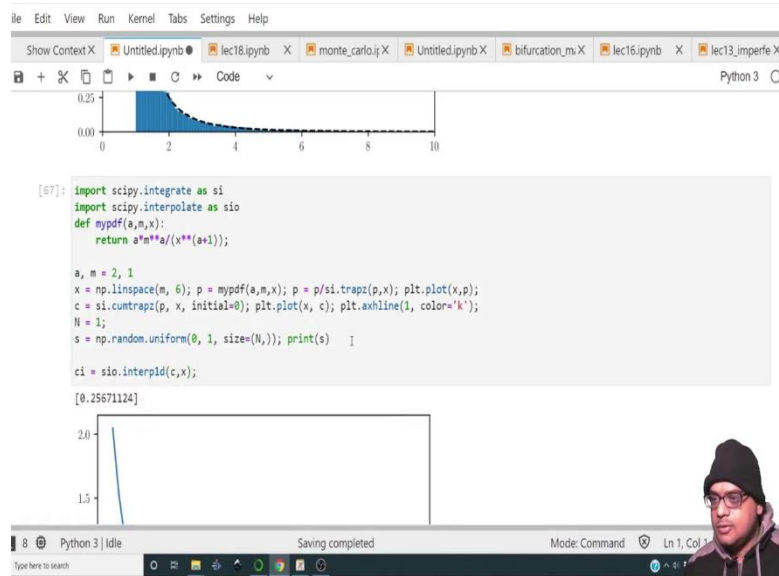
A diagram below the equations shows a coordinate system with the x-axis labeled x and the y-axis labeled c . The x-axis has points x_{min} and x_{max} marked. The y-axis has points 0 and 1 marked. A curve representing the CDF is shown, starting at $(x_{min}, 0)$ and ending at $(x_{max}, 1)$. A point s is marked on the curve, and a vertical line is drawn from s down to the x-axis, indicating the corresponding value of x .

So, let s be `np.random.uniform` between 0 and 1 and let me draw only 1 value. In fact, let me declare it as N , where N will be equal to 1 for now alright. So, `size = N`, right. So, N , means it is just a 1D array. So, let me print out that value. So, s is now 0.16418 ok. So, we have something like 0.16 over here.

So, now it maps onto this particular value in the CDF. I must find out as to for what value of x does that CDF occur. So, how do I do that? So, now, I have what? A bunch of values of x and corresponding to that I have a bunch of values of the CDF. So, X_{min} the value of C will be 0, X_{max} the value of C will be 1.

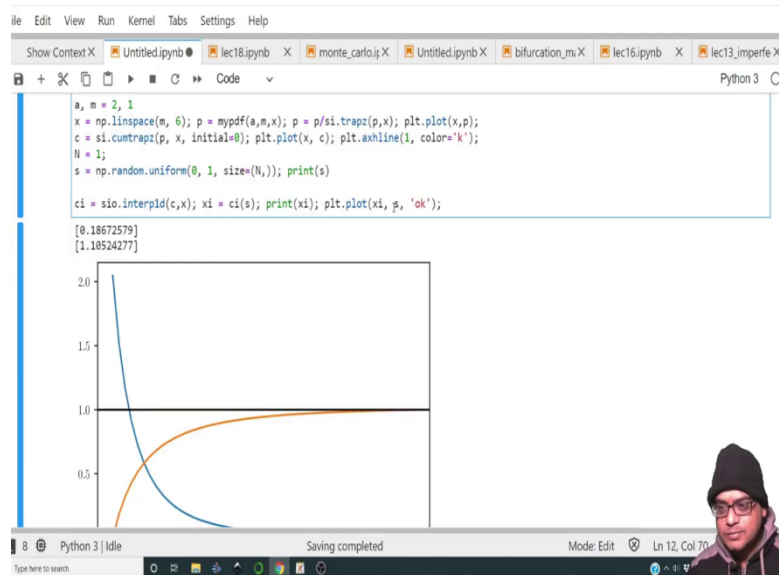
I must make an interpolation between the sampled value so that given the location of the sampled value I can interpolate over X and find out for which value of X I will obtain that CDF. So, I must first create an interpolation function.

(Refer Slide Time: 41:35)



So, for that, I must import `scipy.interpolate` as `sio`. Then I will make `c` interpolate equal to `interpld` and I will make an interpolation function of `c` as a function of `x`, alright. So, so far so good. Now, I will so, what is `ci`? `ci` is now some kind of a interpolant I must pass the value of `s` to the interpolant alright.

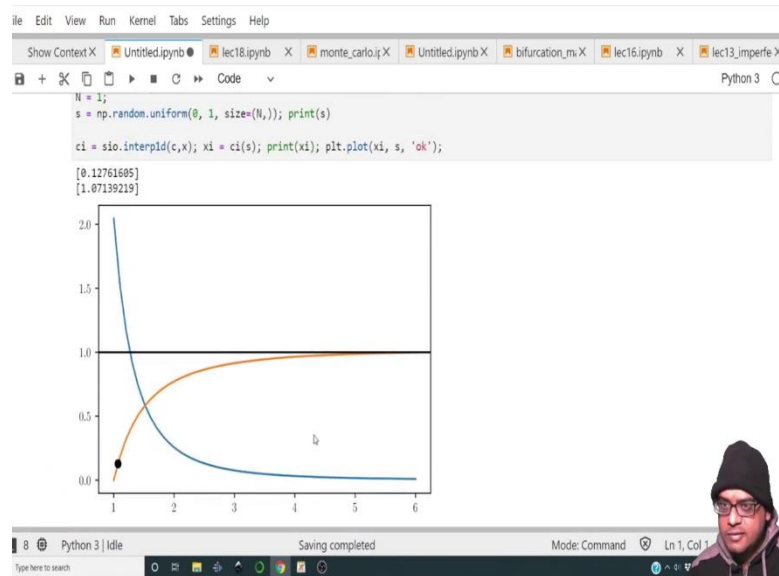
(Refer Slide Time: 42:18)



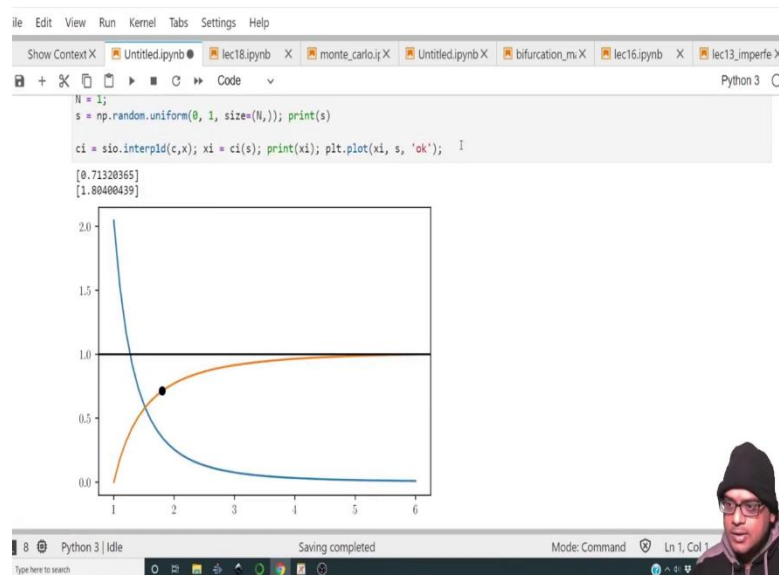
So, I will say `x` interpolant equal to `ci` of `s`. Let me print what `xi` is going to be as well. So, `xi` appears to be this actually each time I run the cell it is going to generate a new random number. But, let us do this. Let us plot `plt dot plot` I will plot(`xi` , `s`) and I will put a black mark.

So, what does it mean? It will show us this corresponding point. So, if this is the randomly value random sample s . It will tell me for which oh sorry, this is not this is the CDF. So, if I have this as the random value s , it will show this as the point which corresponds to x interpolate comma s , alright. So, for that random number between 0 and 1 which has been uniformly sampled this is the corresponding x . So, let us see.

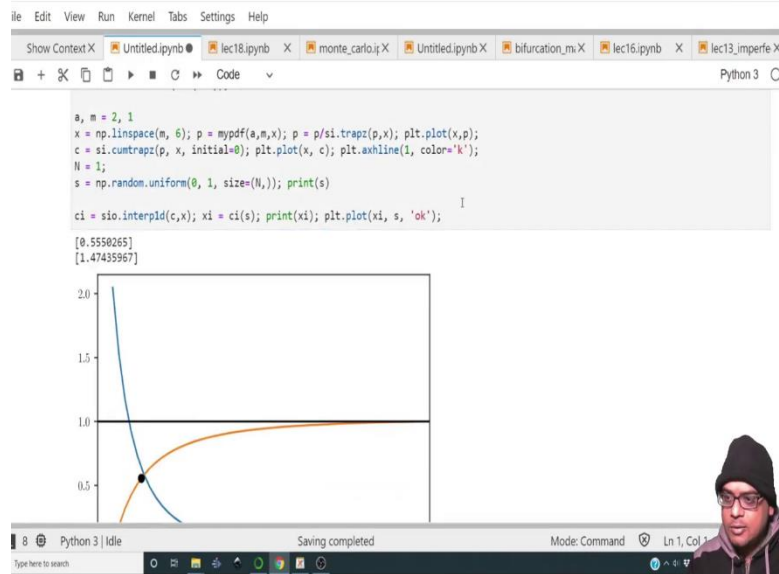
(Refer Slide Time: 43:32)



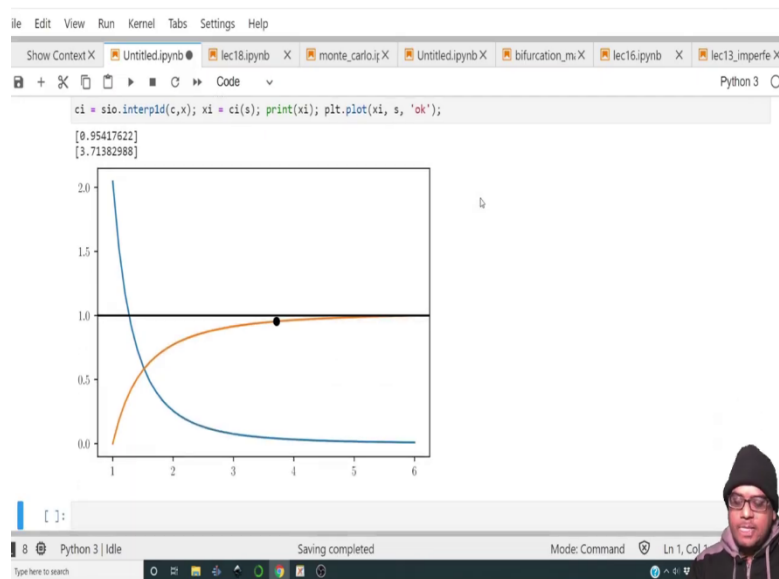
Great, so it lies on the CDF as expected. (Refer Slide Time: 43:35)



(Refer Slide Time: 43:39)

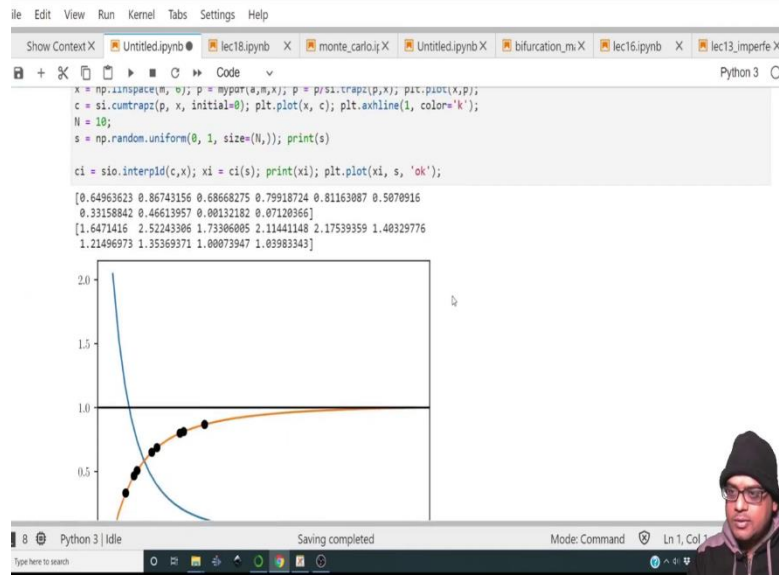


(Refer Slide Time: 43:41)



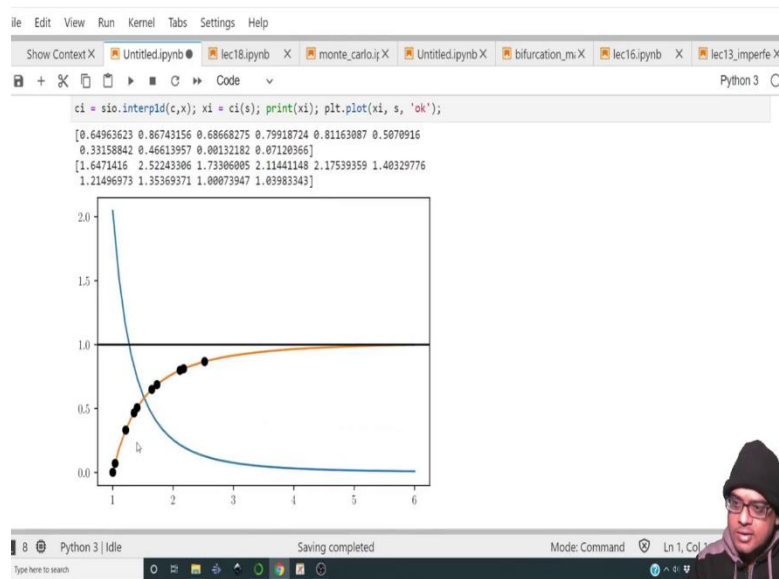
So, we can run this program a bunch of times and we will obtain a different random numbers, alright and now let me sample more number.

(Refer Slide Time: 43:49)



So, let me sample 10.

(Refer Slide Time: 43:51)

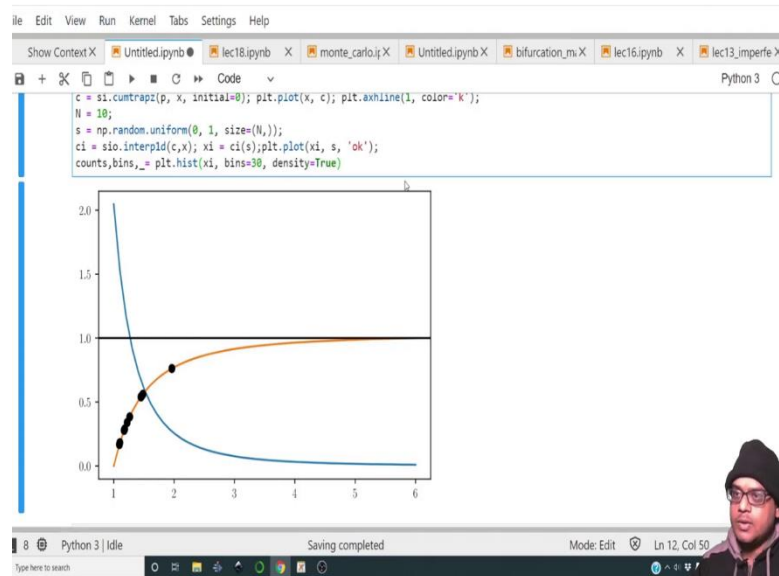


So, when I do this I get 10 values on top of that. So, the values of xi that I get through this inversion process alright the values of xi that I obtain are the samples from that PDF that we wish. That PDF need not be a Pareto redistribution, we can have anything we like.

But, that these are now the xi's that we have sampled and how do I know that the xi's that I have sampled fall under that distribution. We have not shown the mathematical proof this is not a course where I am going to show you the proof as to this method how it helps us to sample values from the required PDF ok. What we expect now is if I draw the

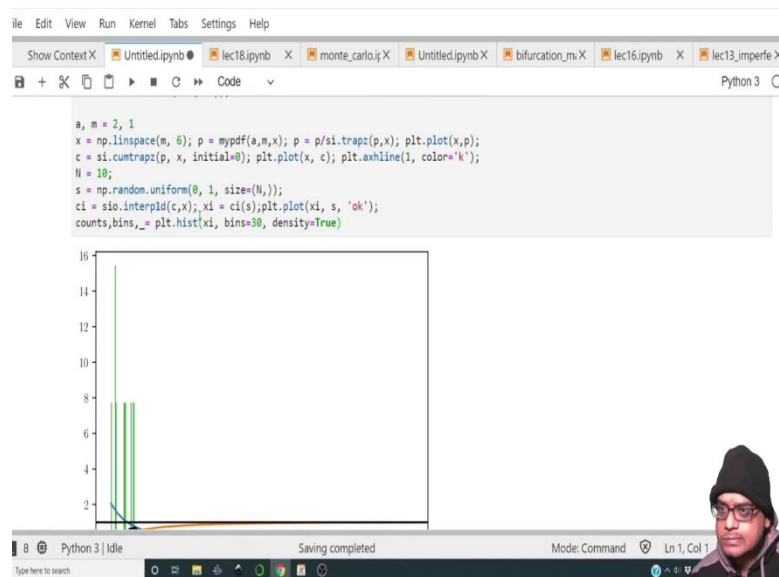
histogram or the density histogram of xi over this range I should obtain this PDF that I have. Let us see whether that is true or not.

(Refer Slide Time: 44:57)



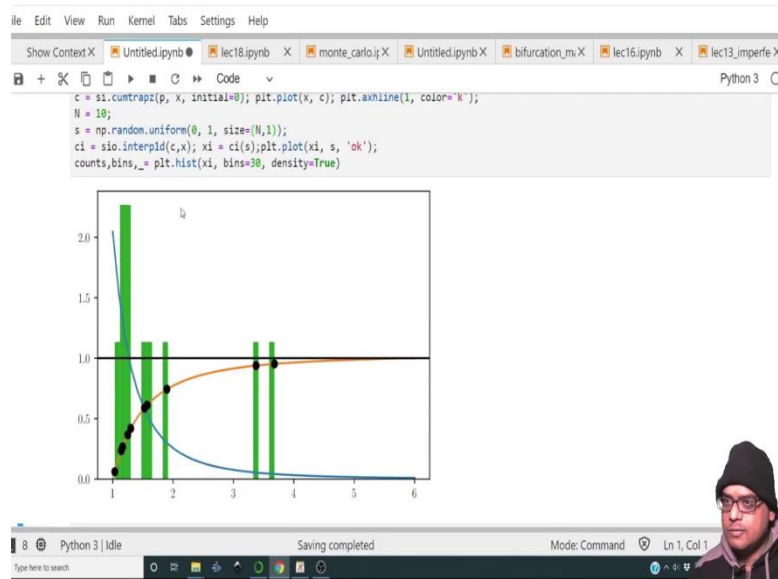
So, let me now suppress the value of s and xi because so, let me just keep the plot. So, now, I must draw the histogram of xi. So, counts ,bins , patches = plt.hist I must pass xi. Let me take bins as 30, density equal to true, let me see.

(Refer Slide Time: 45:26)



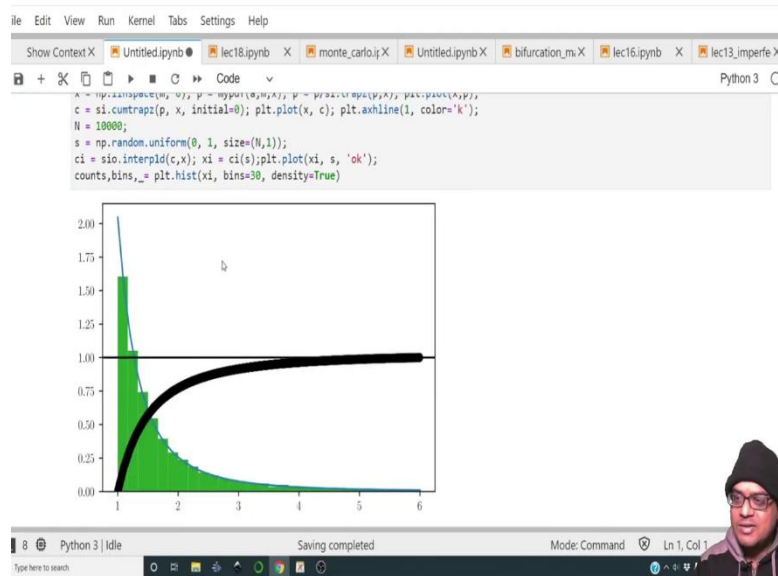
So, it was an issue of this yeah.

(Refer Slide Time: 45:38)



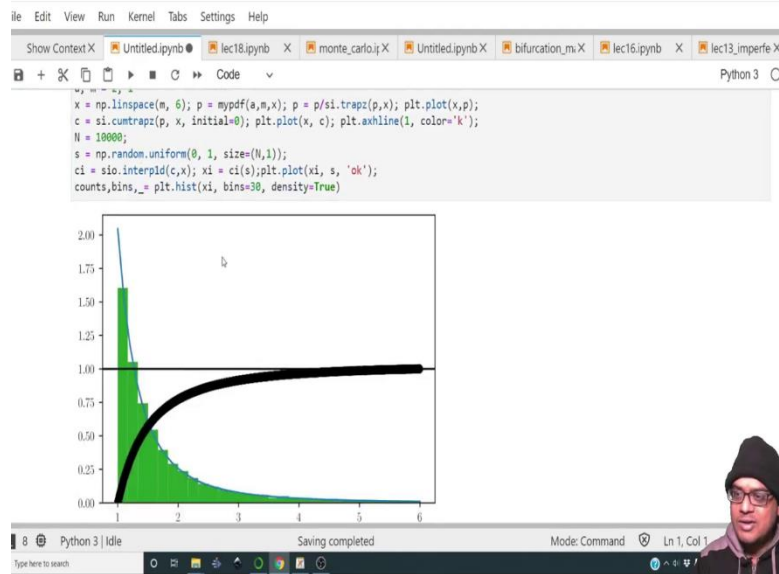
It was an issue of this. So, I must declare it as 1 this small, it is just saying that it has N rows and only 1 column ok.

(Refer Slide Time: 45:51)



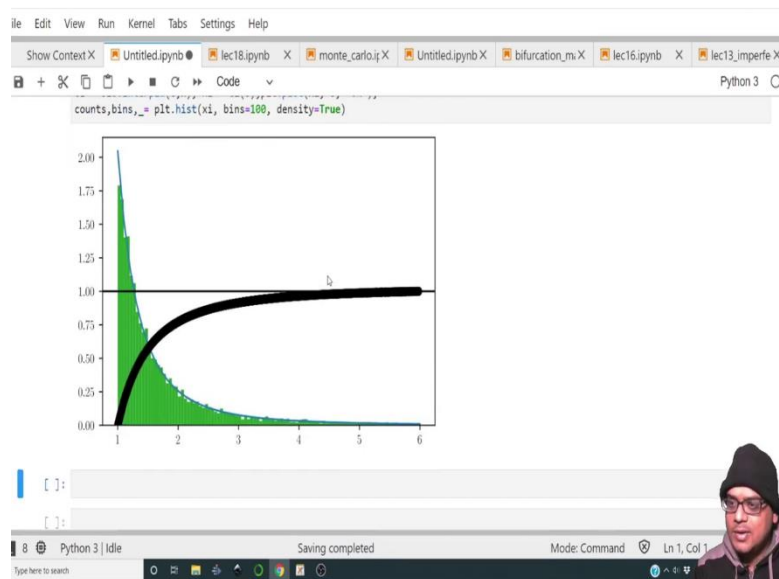
So, let me take 100 or let us say a 100. Great, we do approach the distribution ok.

(Refer Slide Time: 46:00)



If I increase the number further the more number of points I take, I get a better fit to the distribution. Let me increase the number of bins to 100 excellent.

(Refer Slide Time: 46:06)



So, through this technique we are able to sort of interpolate a uniformly sampled variable with the help of the CDF to find out a random variable from the PDF that we desire. We cannot just give a PDF and we cannot sample directly from that PDF. We must go through this approach in order to sample from the desired PDF ok.

Think about this, go through the mathematics if you are interested, but in general if you are if you are not so bothered about creating your own probability density functions we

can use the host of the distributions that are available in NumPy or SciPy and get your job done. You do not need to do all this alright.

So, in the next class we will continue with this lecture forward and find out how we can sort of estimate pi with the help of a Monte Carlo simulation from the dart throwing problem to the Buffon needle problem. With this I am ending this particular lecture and I will see you again next time.

Bye.