



# REVIEW\* OF STATISTICS AND DISTRIBUTIONS

Note: the examples in this lecture can be found in [Files/Jupyter Notebooks/Lecture 18.ipynb](#)

---

Eric Prebys  
)j  
Winter 2024

\* I hope!



# Probabilities

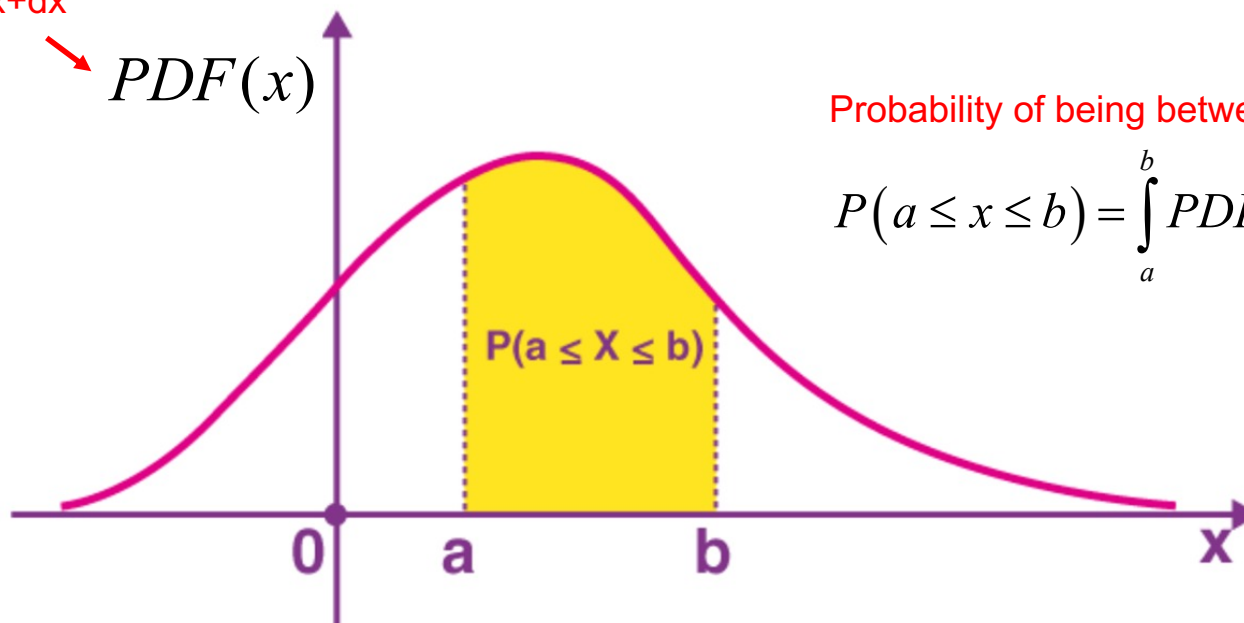
- At the microscopic level, everything in physics is governed by probabilities.
- When statistics become large enough, the uncertainties become small enough that probable outcomes become virtual certainties
  - This is how casinos make money!
- In this course, we will analyze some of the statistical properties most often encountered in physics.



# Probability Distributions

- In general, we can characterize a probability by a “Probability Distribution Density Function” (PDF). In one dimension, this is defined as

Probability of being  
between  $x$  and  $x+dx$



Probability of being between  $a$  and  $b$  is

$$P(a \leq x \leq b) = \int_a^b PDF(x) dx$$

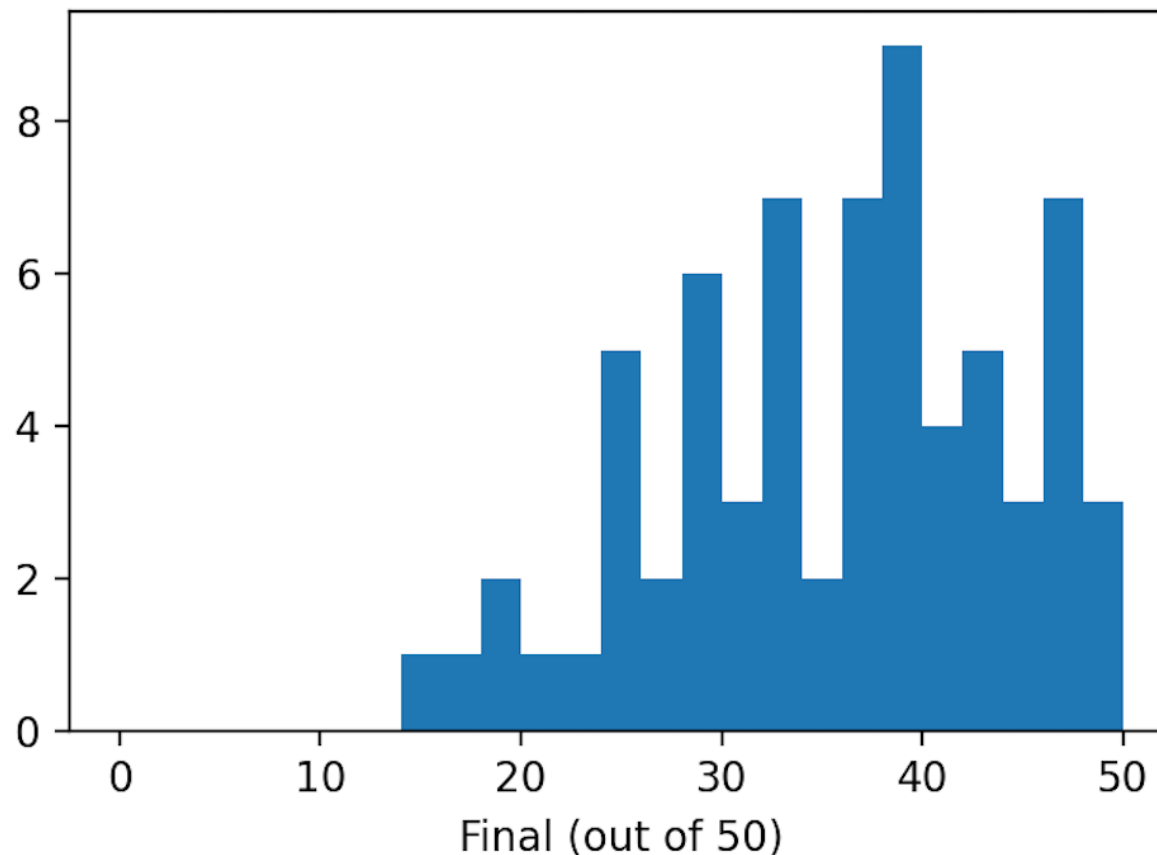
Normalization:

$$\int_{-\infty}^{\infty} PDF(x) dx = 1$$



# Histograms

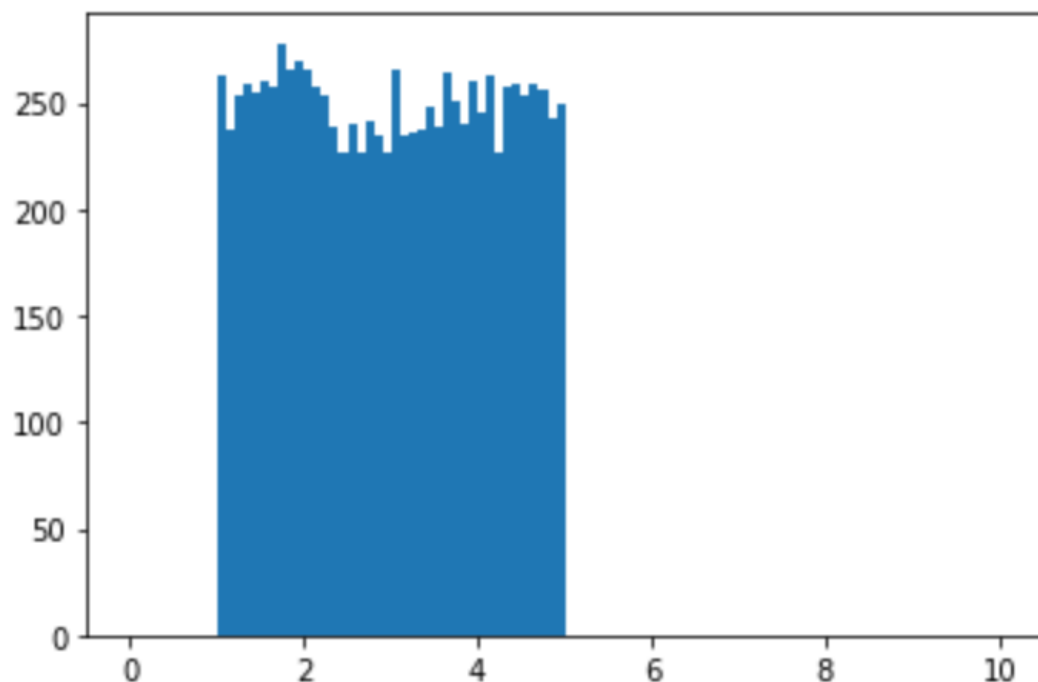
- We generally plot distributions using “histograms”, which record the number of entries falling within particular ranges.
  - Example: score on Phy 009D final exam





# Uniform Probability

- A uniform probability is a distribution within a specified range in which every value in the range has an equal probability.
- Example: a uniform distribution from 1 to 5



$$x < x_{min} : PDF(x) = 0$$

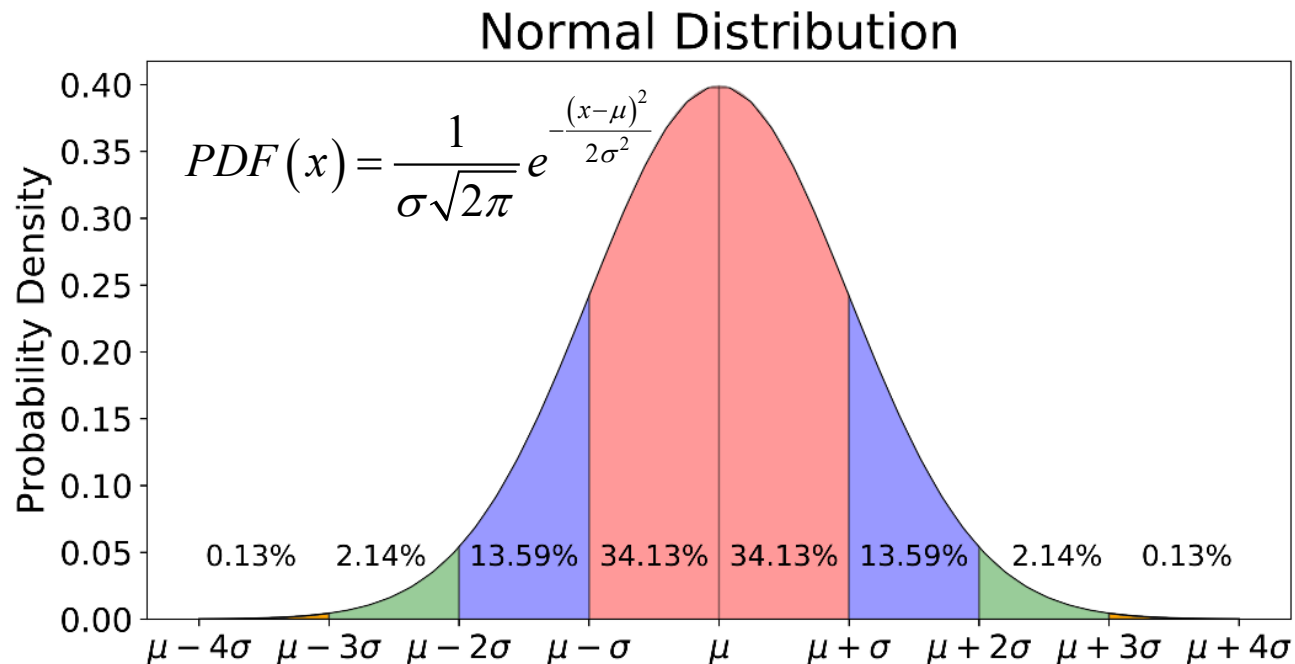
$$x_{min} \leq x \leq x_{max} : PDF(x) = \frac{1}{(x_{max} - x_{min})}$$

$$x > x_{max} : PDF(x) = 0$$



# Normal Distribution

- A normal (or “Gaussian”) distribution is a probability characterized by a mean ( $\mu$ ) and a standard deviation ( $\sigma$ ), according to the following PDF



- As we will see, in some limit, all errors can be characterized by this distribution (“central value theorem”)



# Random Numbers in Python

- The `numpy.random` package contains a powerful set of routines to generate random numbers, including
  - Uniform random numbers in a specified range
  - Distributions based on user-specified probabilities
  - Normal (Gaussian) distributions
  - Binomial distributions (we'll discuss this shortly)
  - Many, many, more
- Without a physical inputs, computers cannot generate truly random numbers. They generate “pseudo-random” numbers, based on a starting “seed”.
  - If unspecified, they usually use the unique time as the seed.
  - If a seed is specified (`np.seed(int)`), then *identical* subsequent random number calls will produce identical numbers each time
    - This can be very important for debugging code!
- Do uniform and Gaussian demo...



# Binomial Distributions

- Let's imagine I flip a coin 10 times, and it comes up heads 4.
- If I look at the first heads, there are 10 possibilities for which throw it came from, 9 for the second, 8, for the third, and 7 for the fourth, for a total of

$$10 \times 9 \times 8 \times 7 = \frac{10!}{6!} = \frac{10!}{(10-4)!}$$

possibilities

- But I don't actually care which throw each came for, so for the first heads there are 4 possibilities that are identical, 3 for the second, the for a total of 4! Indistinguishable "permutations", so the total number of cases with 4 heads is

$$\frac{10!}{4!(10-4)!}$$

- Generalizing this, the number of times, we get  $n_H$  out of  $N$  throws is given by

$$W_{n_H}^N = \frac{N!}{n_H!(N-n_h)!} \equiv \binom{N}{n_H} \quad \text{"Binomial coefficient"}$$





# Probabilities

- If I flip a coin  $N$  times, there are  $2^N$  possible unique sequences, so the probability of finding  $n_H$  after  $N$  throws is

$$P_{n_H}^N = \frac{1}{2^N} \frac{N!}{n_H!(N-n_h)!} \equiv \frac{1}{2^N} \binom{N}{n_H}$$

- For 10 throws, there are  $2^{10}=1024$  possible combinations, so the table of probabilities is

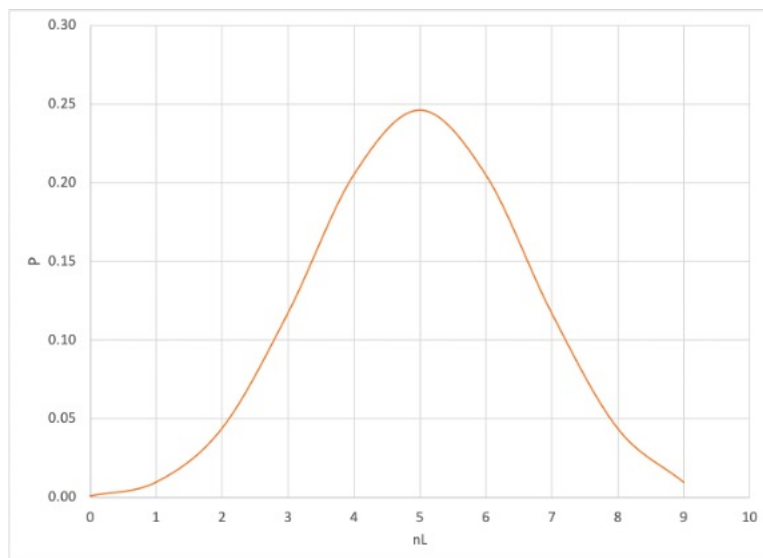
$n_H$	$W(10, n_H)$	$P(n_H)$
0	1	0.0010
1	10	0.0098
2	45	0.0439
3	120	0.1172
4	210	0.2051
5	252	0.2461
6	210	0.2051
7	120	0.1172
8	45	0.0439
9	10	0.0098
10	1	0.0010



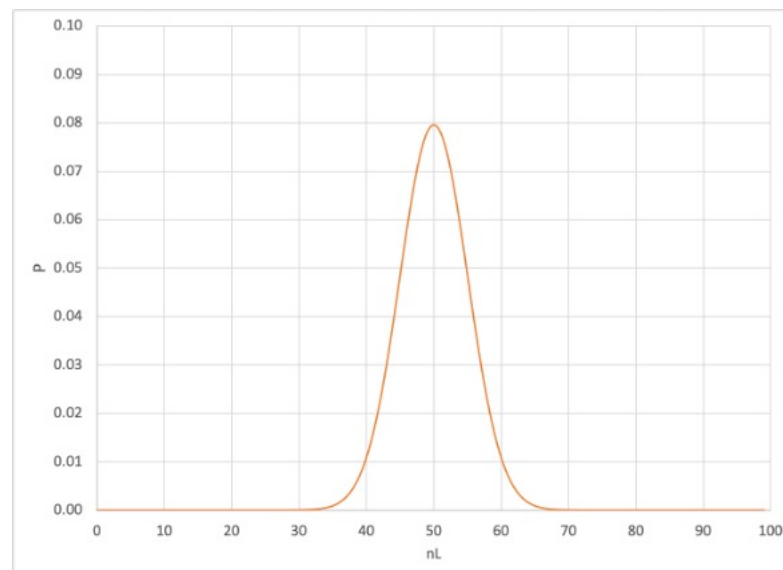
# Going to More Throws...

- The more throws, the narrower the distribution becomes

10 Throws



100 Throws



- Of course, the number of particles in a box is much bigger than 100, so the fraction on the left will be very close to  $\frac{1}{2}$ , but will have a distribution.



# Generalized Binomial Distribution

- In our example, a coin had an equal probability (50%) of coming up heads or tails.
- If instead we have two distinct outcomes, **A** and **B**, in which the probability of **A** is  $p$ , the probability of **B** is therefore  $(1-p)$ , and the normalized binomial distribution is modified as follows
  - The probability of  $n_A$  outcomes in state A out of  $N$  total trials is

$$P_{n_A}^N = \frac{N!}{n_A! (N - n_A)!} p^{n_A} (1 - p)^{(N - n_A)} \equiv \binom{N}{n_A} p^{n_A} (1 - p)^{(N - n_A)}$$

- The average value over many trials of  $N$  “throws” will be

$$\lambda = pN$$

- And the RMS of the distribution will be

Relative width gets narrower

$$\sigma = \sqrt{Np(1 - p)} = \sqrt{\lambda(1 - p)} \quad \frac{\sigma}{\lambda} = \sqrt{\frac{(1 - p)}{\lambda}} = \sqrt{\frac{(1 - p)}{Np}}$$



# Simulating Binomial Distributions in Python

- We will simulate binomial distributions with
  - `numpy.random.binomial(N,p,Ntrial)`

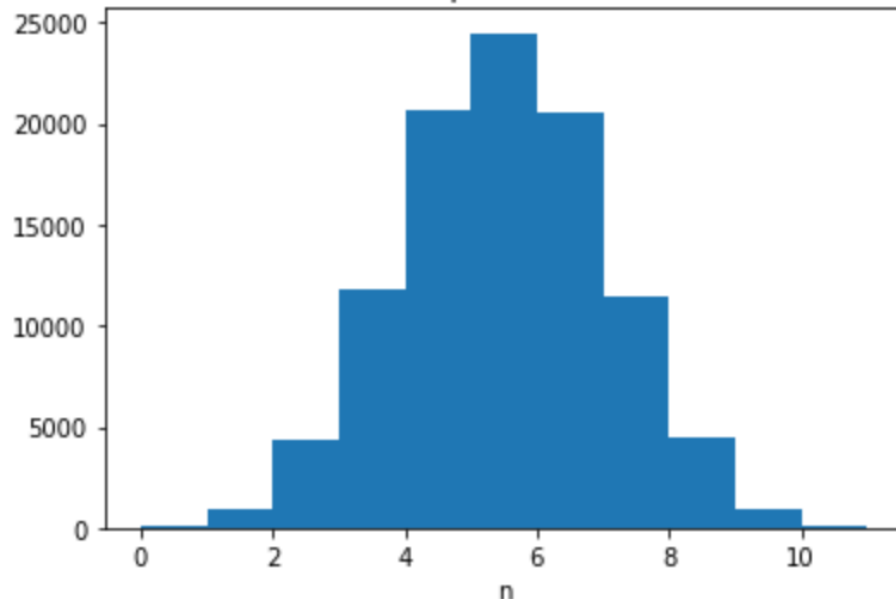
Number of “throws”

Probability each throw

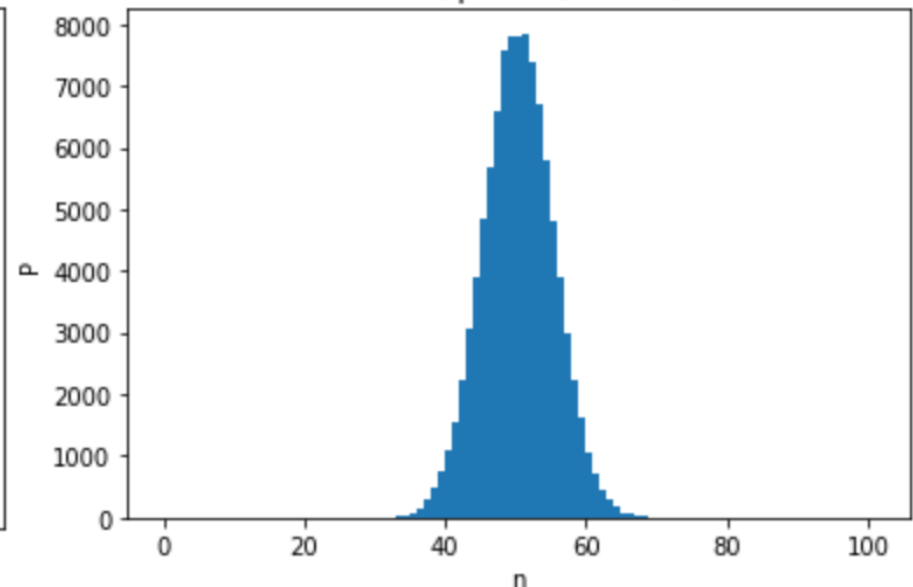
Number of times I repeat the test.  
If 1 (or missing), it returns a scalar,  
otherwise it returns a vector

- For a coin,  $p=1/2$

Binomial Distribution,  $p=0.50$ ,  $N=10$ ,  $ntrial=100000$



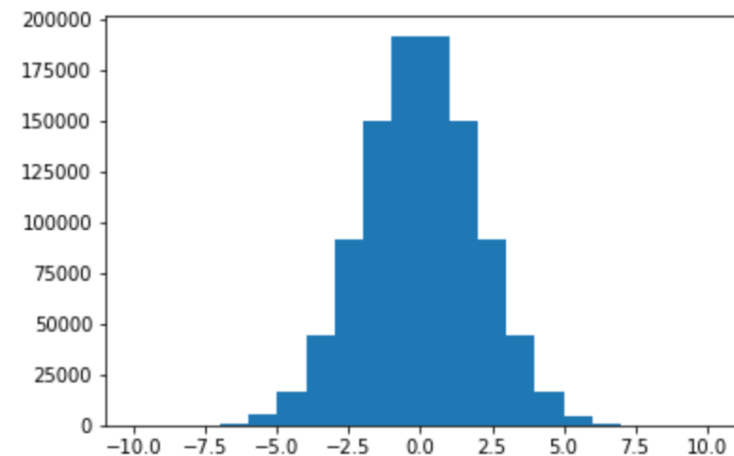
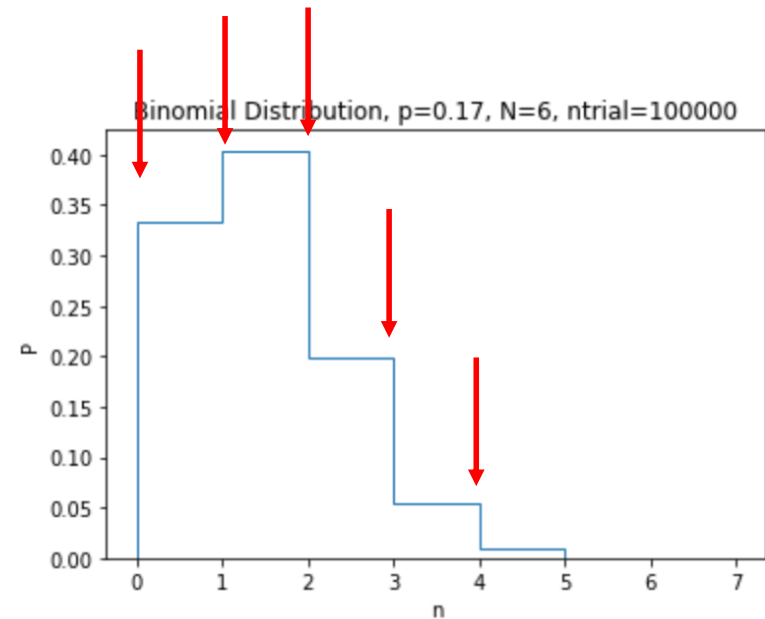
Binomial Distribution,  $p=0.50$ ,  $N=100$ ,  $ntrial=100000$





# Location of the Bin

- In most of the examples we've done, we've been histogramming *integers* into *unit* bins, so the location of our entries is always the left side of the bin, which we get with `bins[:-1]` (the last entry in `bins` is the final right location)
- On the other hand, the normal distribution is a continuous variable, in which the values are distributed ~uniformly over the bin.
  - Makes more sense to put the location in the center of the bin.



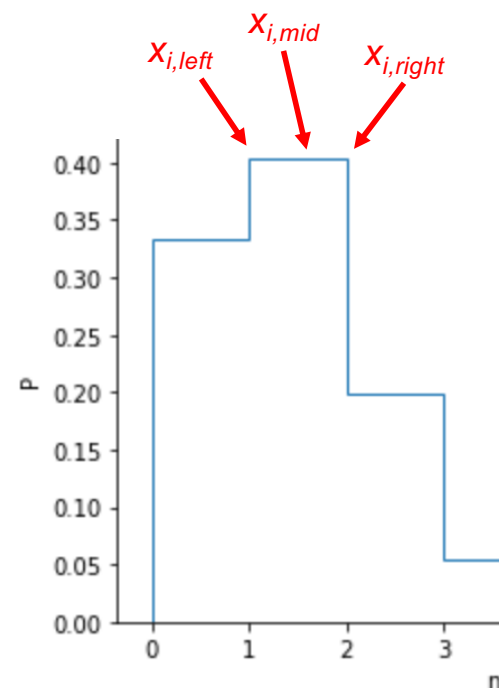


# Fitting to Functions

- The number of entries in a particular bin of a histogram is given by integrating the PDF over the width of the bin.
- We can generally approximate this by multiplying the central value times the bin width

$$N_i = \int_{x_{i,left}}^{x_{i,right}} PDF(x) dx \approx (x_{i,right} - x_{i,left}) x_{i,mid}$$

- This is another good argument for using the center of the bin, which is weirdly done with the “align=‘left’” option.





## Die Example...

- If  $p$  is the probability of an event occurring, then the expected number of events for  $N$  “throws” is

$$\lambda = pN$$



- But what if  $\lambda < 1$ ?
- Example: throwing a 6-sided die three times
- The chance of it coming of any particular number, say 4, is  $1/6$ , so the average number of will come up 6 in 3 throw is

$$\lambda = \frac{1}{6} 3 = .5$$

- Of course, it can't come up .5 times, it can only come up 0, 1, 2, or 3 times.



# Binomial Distribution for Die Throws

- In this case, our options are coming up 4 ( $p=1/6$ ) or coming up any other number ( $q=(1-p)=5/6$ )
- The binomial theorem tells us that the probability of coming with 4 a particular number of times is given by

$$P_{n_A}^N = \binom{N}{n_A} p^{n_A} (1-p)^{(N-n_A)} \rightarrow P_{n_4}^3 = \frac{3!}{n_4! (3-n_4)!} \left(\frac{1}{6}\right)^{n_4} \left(\frac{5}{6}\right)^{(3-n_4)}$$

$$P_0^3 = \frac{3!}{0!(3)!} \left(\frac{1}{6}\right)^0 \left(\frac{5}{6}\right)^3 = 0.5787$$

$$P_1^3 = \frac{3!}{1!(2)!} \left(\frac{1}{6}\right)^1 \left(\frac{5}{6}\right)^2 = 0.3472$$

$$P_2^3 = \frac{3!}{2!(1)!} \left(\frac{1}{6}\right)^2 \left(\frac{5}{6}\right)^1 = 0.0694$$

$$P_3^3 = \frac{3!}{3!(0)!} \left(\frac{1}{6}\right)^3 \left(\frac{5}{6}\right)^0 = 0.0046$$

$$P_0^3 + P_1^3 + P_2^3 + P_3^3 = 1$$

Hist values = [0.577866 0.347917 0.06961 0.004607]

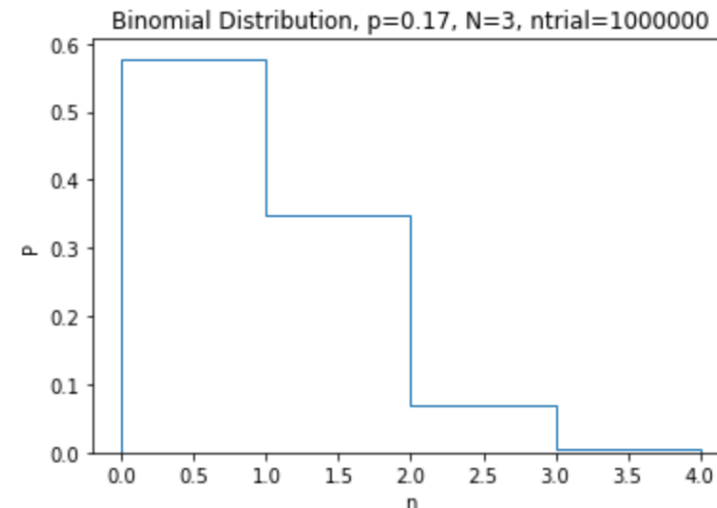
Predicted average = 0.5000

Measured average = 0.5010

Predicted standard deviation = 0.6455

Measured standard deviation = 0.6456

Standard deviation/N = 0.21521593348593457







# Approximations

- The binomial expression is exact, but factorials get problematic pretty fast

N	FACT(N)
168	2.5261E+302
169	4.2691E+304
170	7.2574E+306
171	#NUM!

170! is the biggest number that can fit in a 64-bit floating point, and we deal with numbers in the millions, billions, or more

- Generally, we use two approximations, depending on the expectation values on the counts
  - “Small statistics” ( $\lambda < 10$ ): Poisson statistics
    - Example: throwing a die 3 times
  - “Large statistics” ( $\lambda \geq 10$ ): Gaussian statistics
    - Example: flipping a coin at least 10 times



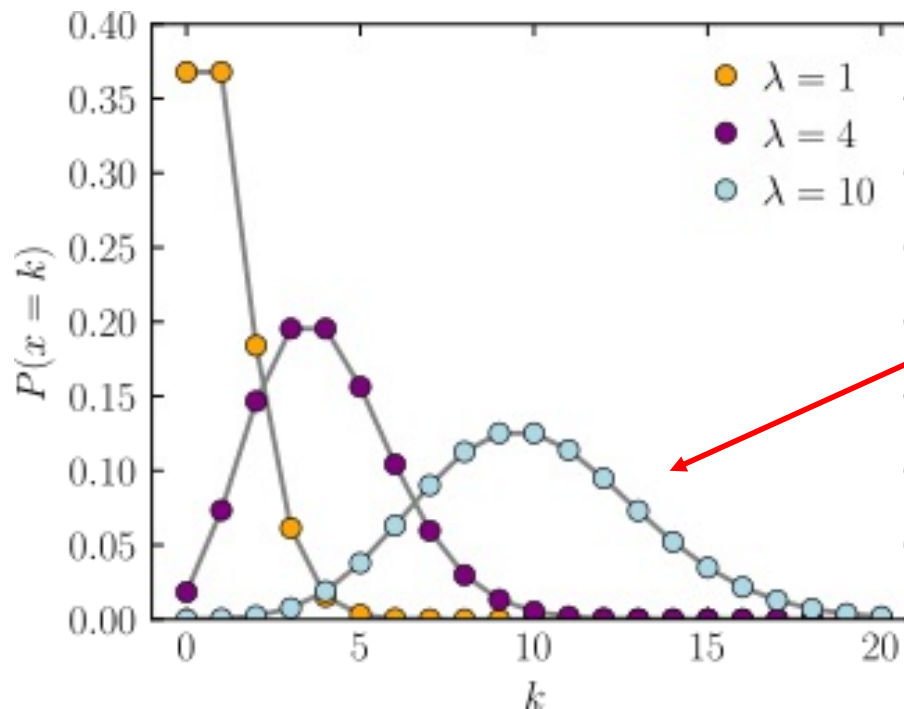
# Poisson Distribution

Probability *Mass* Function

- For small statistics ( $\lambda < 10$ ), the PMF can be approximated by

$$P(k) = \frac{1}{k!} \lambda^k e^{-\lambda}$$

This equation will also continue to work, but again, these terms will quickly become problematic for large  $\lambda$



Note the equation becomes symmetric for a  $\lambda$  of 10 or more

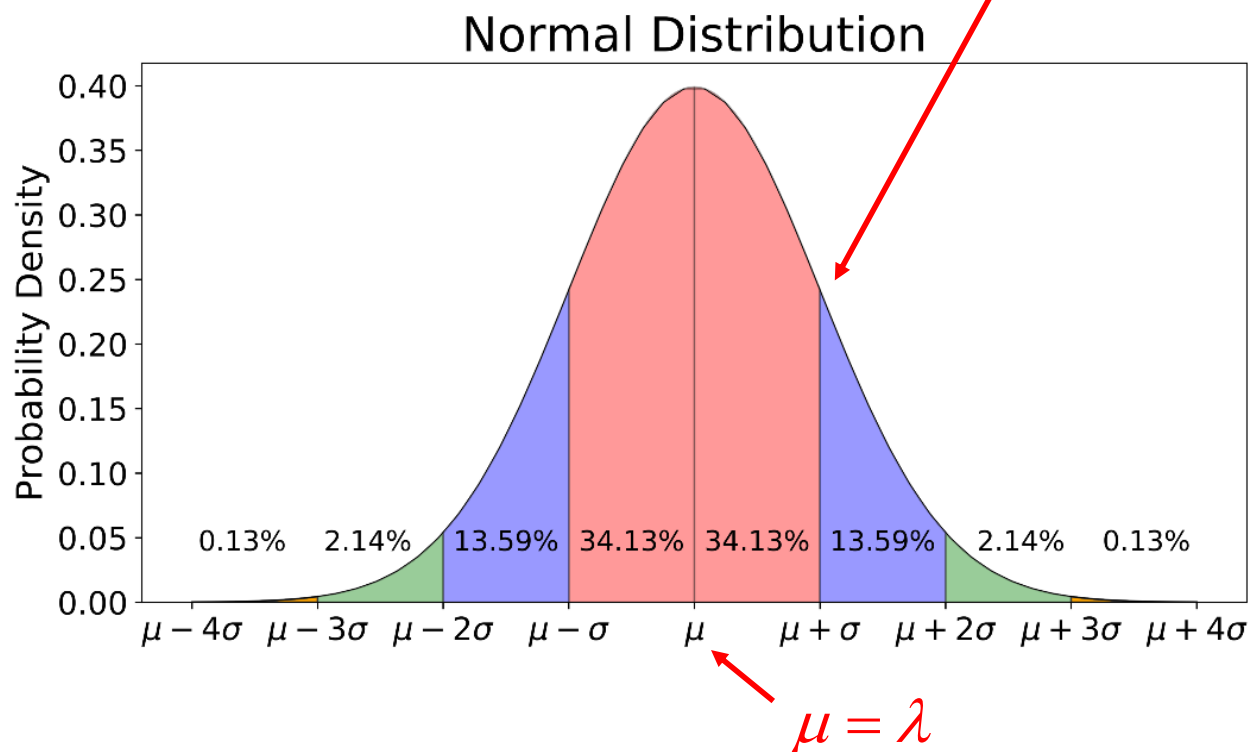


# Gaussian Distribution

- For larger values of  $\lambda$ , we can approximate things with a gaussian distribution.

$$P(k) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(k-\lambda)^2}{2\sigma^2}}$$

$\sigma = \sqrt{\lambda(1-p)}$





# Generating These Distributions Functions in Python

- We will use the `scipy.stats` package

- Poisson distribution

- `scipy.stats.poisson.pmf(k,lam)`

Point or array of points

“probability mass function”

$\lambda$

- PMF designed for discrete integer points

- Gaussian (normal) distribution

- `scipy.stats.norm.pdf(x,loc=mean,scale=sigma)`

Point or array of points

“probability density function”

$\lambda$

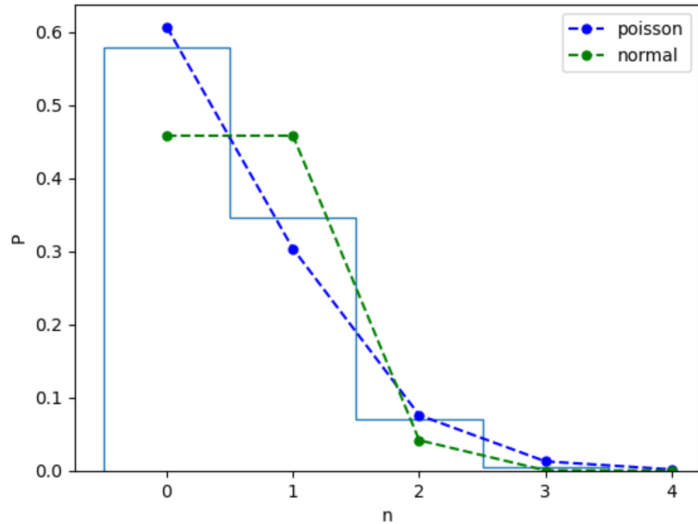
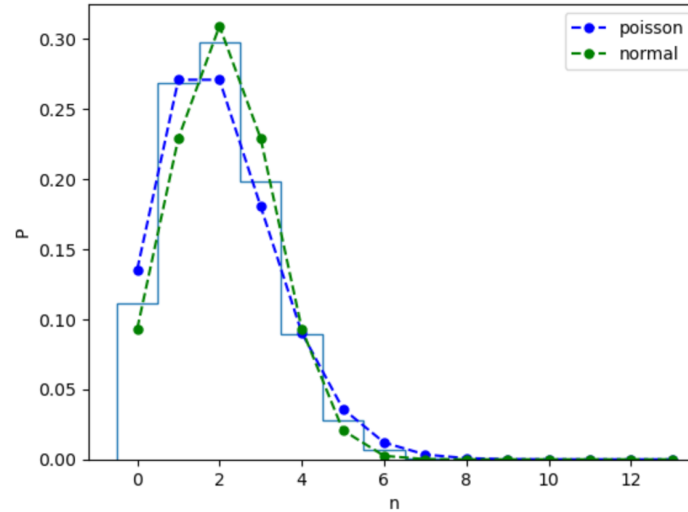
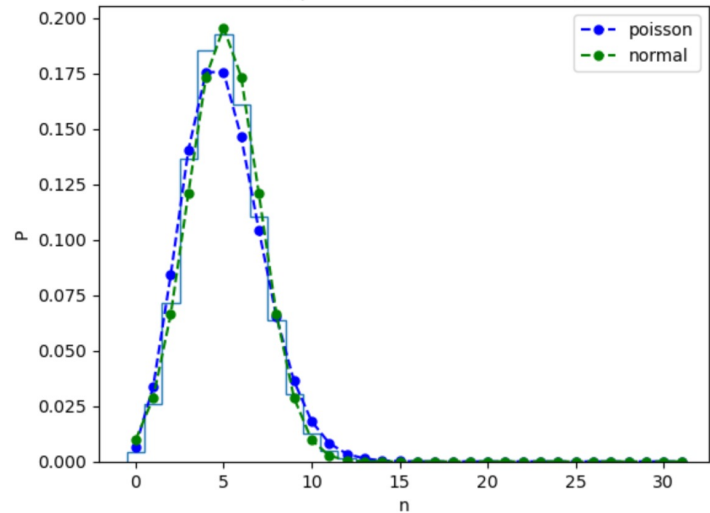
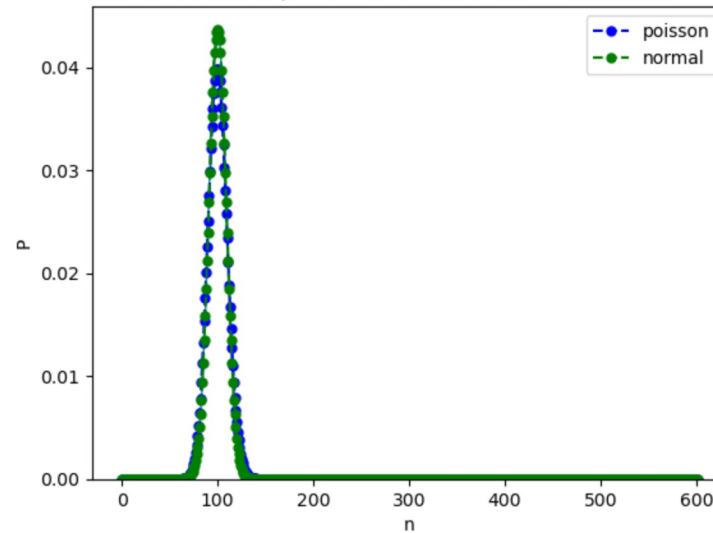
$\sigma$

- PDF is a continuous function

- *strictly speaking*, we should integrate it over each integer value, but we usually won't do that



# Back to Dice

Binomial Distribution,  $p=0.1667$ ,  $N=3$ ,  $\text{lam}=0.5$ ,  $\text{ntrial}=100000$ Binomial Distribution,  $p=0.1667$ ,  $N=12$ ,  $\text{lam}=2.0$ ,  $\text{ntrial}=100000$ Binomial Distribution,  $p=0.1667$ ,  $N=30$ ,  $\text{lam}=5.0$ ,  $\text{ntrial}=100000$ Binomial Distribution,  $p=0.1667$ ,  $N=600$ ,  $\text{lam}=100.0$ ,  $\text{ntrial}=100000$ 

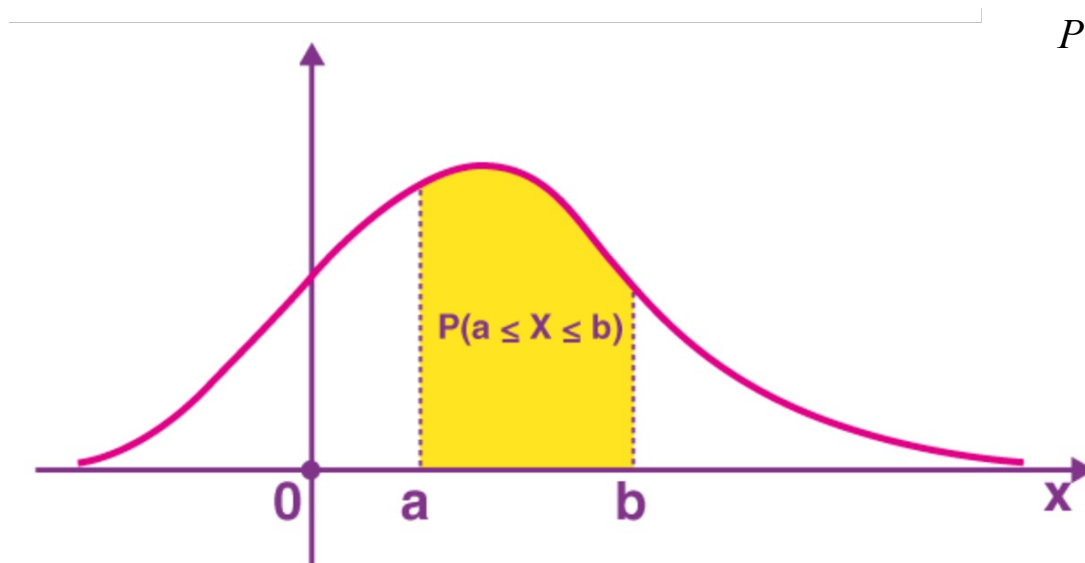


# Cumulative Density (or Mass) Functions

- For any probability distribution, we can define the ``Cumulative Density function as the integral up to a particular value.

$$CDF(x) = \int_{-\infty}^x PDF(z) dz$$

- So we can get the probability of falling between any two points as



$$\begin{aligned} P &= \int_a^b PDF(x) dx \\ &= \int_{-\infty}^b PDF(x) dx - \int_{-\infty}^a PDF(x) dx \\ &= CDF(b) - CDF(a) \end{aligned}$$

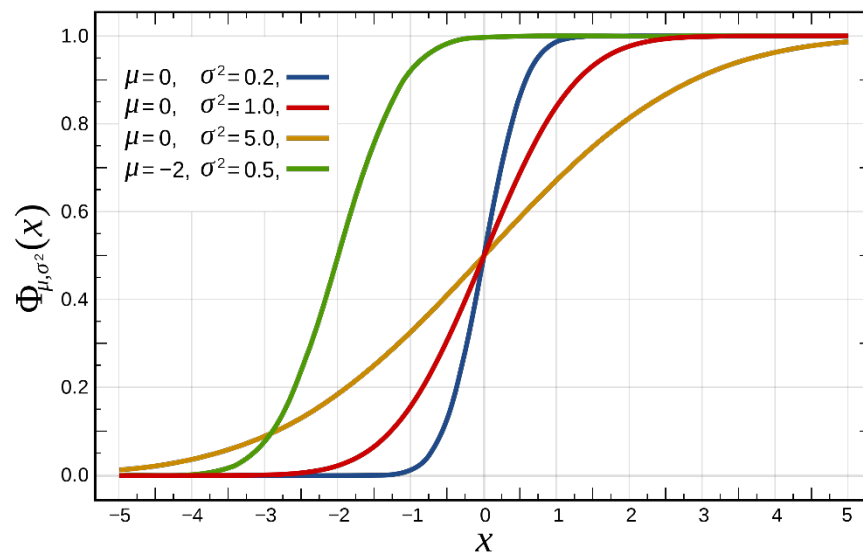
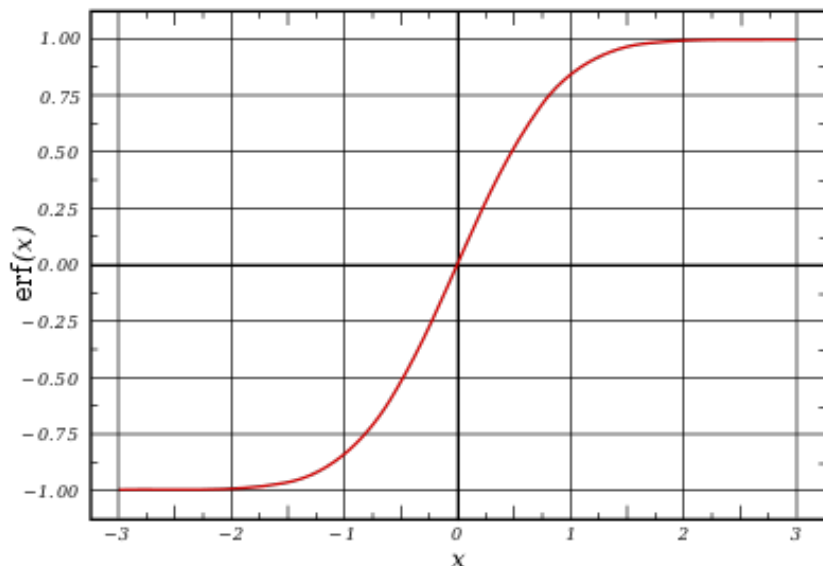


# CDF for a Gaussian

- Unfortunately, the indefinite integral of a Gaussian is not defined, so we define it in terms of the “error function” (erf())

$$\text{erf}(x) \equiv \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

$$\begin{aligned} \text{CDF}_{\text{norm}}(x) &= \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{(z-\mu)^2}{2\sigma^2}} dz \\ &= \frac{1}{2} \left( 1 + \text{erf}\left(\frac{x-\mu}{\sqrt{2}\sigma}\right) \right) \end{aligned}$$





# Gaussian CDF in Python

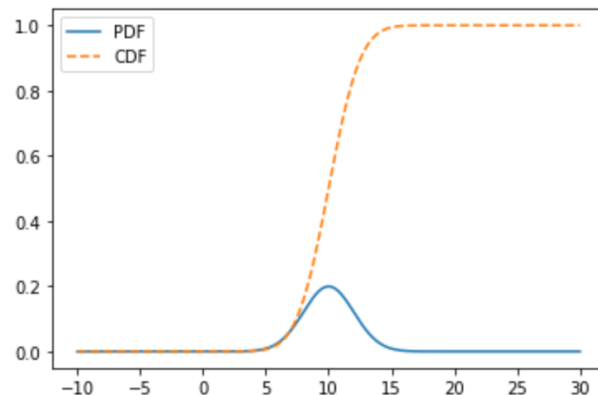
- In Python, the CDF is extracted through the `cdf` method of `stats.norm`

```
[84... # Generate Nsignal events according to a Guassian distribution  
mu = 10.  
sig = 2.  
Nsignal=1000  
ssig = np.random.normal(mu,sig,Nsignal)
```

```
[85... xlin = np.linspace(-10.,30.,1000)
```

```
[86... ypdf = stats.norm.pdf(xlin,loc=mu,scale=sig)
```

```
[88... ycdf = stats.norm.cdf(xlin,loc=mu,scale=sig)  
pl = plt.plot(xlin,ypdf,label='PDF')  
pl = plt.plot(xlin,ycdf,"--",label='CDF')  
leg=plt.legend()
```

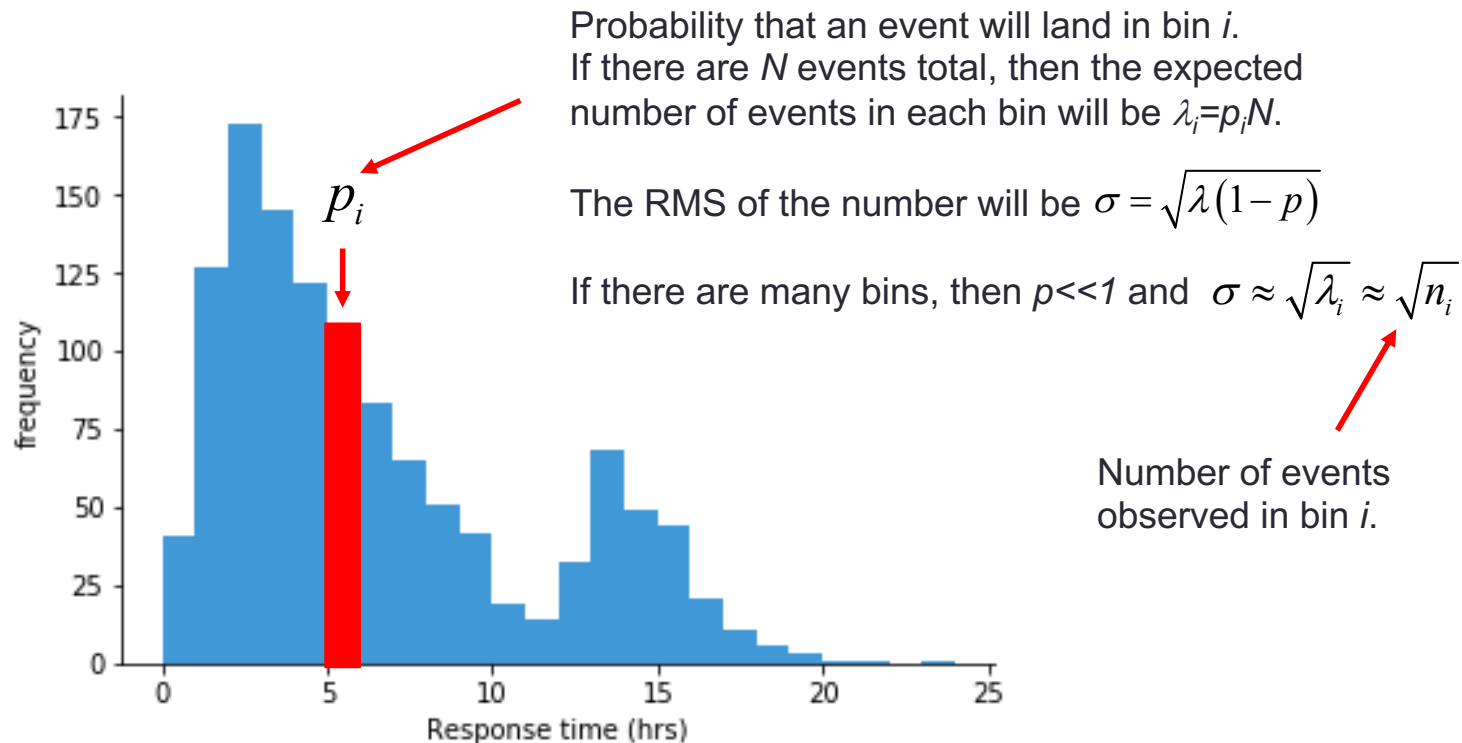






# Error on Histogram Bin Contents

- If entries are distributed over many histogram bins, we can look at any single bin and consider the binary choice between that bin and all other bins.






# Limit of Small Statistics

- We generally set the error on the contents of a bin as the square root on the number of the events observed in the bin.
- This isn't really accurate when the number of events in the bin is small (say  $<5$ ).
- In particular, if there are no events in the bin, this would say

$$\sigma_i \approx \sqrt{n_i} = \sqrt{0} = 0$$

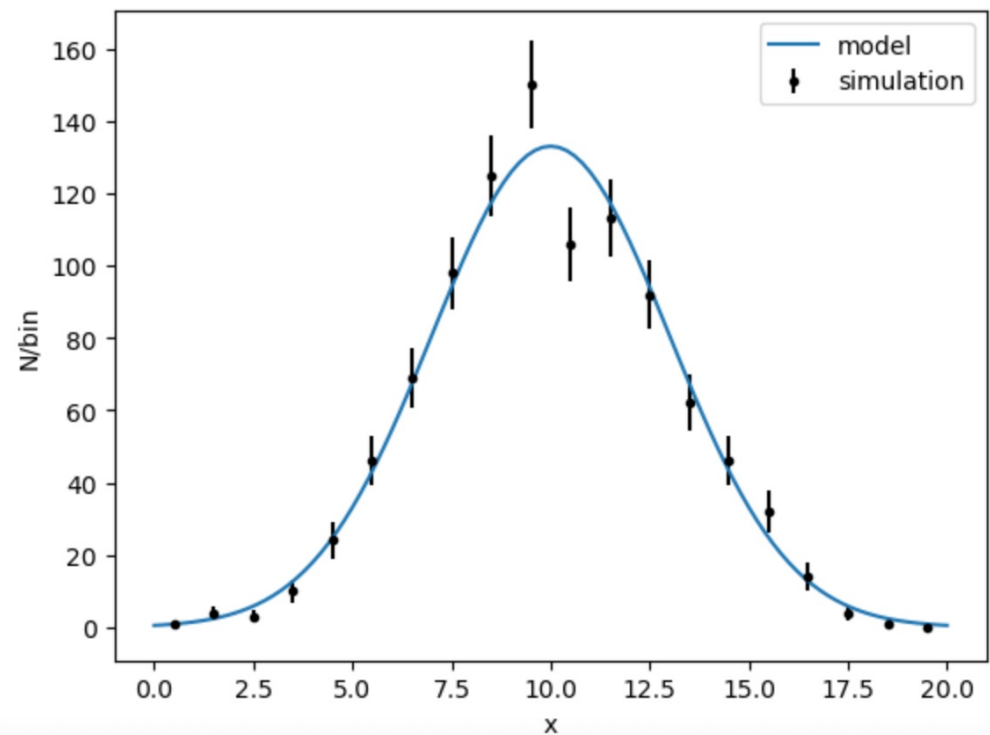
No error on point! 

- This will cause any fit to fail if it can't go exactly through the point.
- If we have a small number of bins with zero, we typically set the error to 1.
- If there are a significant number of bins with zero, we need to use a different approach to fitting.



# Plotting with Errors

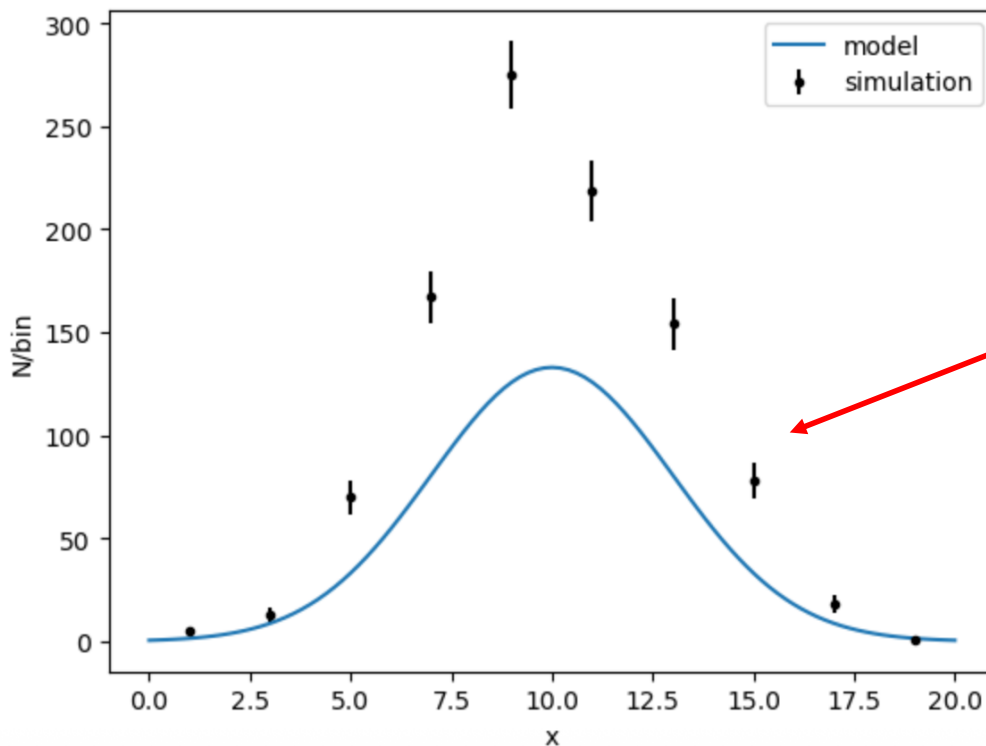
```
# Histogram them with Unit bin sizes
Nbins = 20
hist,bins = np.histogram(ssig,bins=Nbins,range=(0,Nbins))
hist_error = np.sqrt(hist)
hist_error = np.clip(hist_error,1.,None) # If the sqrt is 0, set it to 1.
xcent = .5*(bins[:-1]+bins[1:]) # Center of the bins
plt.errorbar(xcent, hist, hist_error, fmt="k.",
             label="simulation")
xlin = np.linspace(0.,20.,101)
pl=plt.plot(xlin,Nsignal*stats.norm.pdf(xlin,loc=mu,scale=sig),label='model')
plt.xlabel('x')
plt.ylabel('N/bin')
plt.legend()
plt.show()
```





# Normalizing Histograms (Important Concept!)

- In case you haven't noticed, in most of our examples, we've always made our bin size 1. What happens if we change that. Let's redo the last histogram with a bin size of 2



What happened?

We added pairs of bins together, which doubled their contents, but my physics shouldn't depend on how I bin things.

How can I present things in a consistent way?



- Our PDF is defined by the normalization condition

$$\int_{-\infty}^{\infty} PDF(x) dx = 1$$

- The probability of something landing in a particular bin is

$$\int_{x_{lo}}^{x_{hi}} PDF(x) dx = \langle PDF \rangle (x_{hi} - x_{lo})$$

Average value over the bin

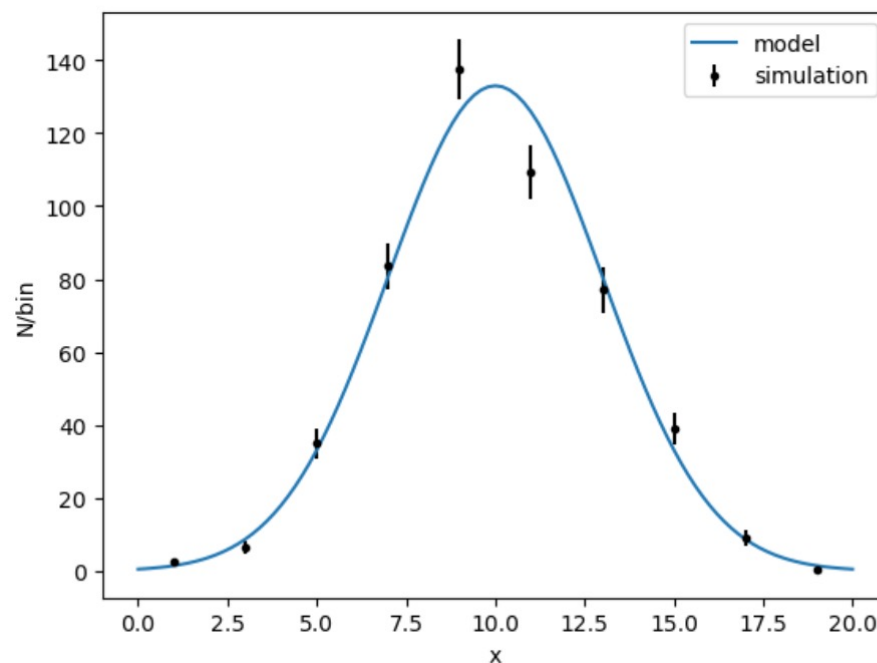
- If I divide the bin contents (and the error!) by the size of the bin, I should get a result that's independent of bin size.



# Check

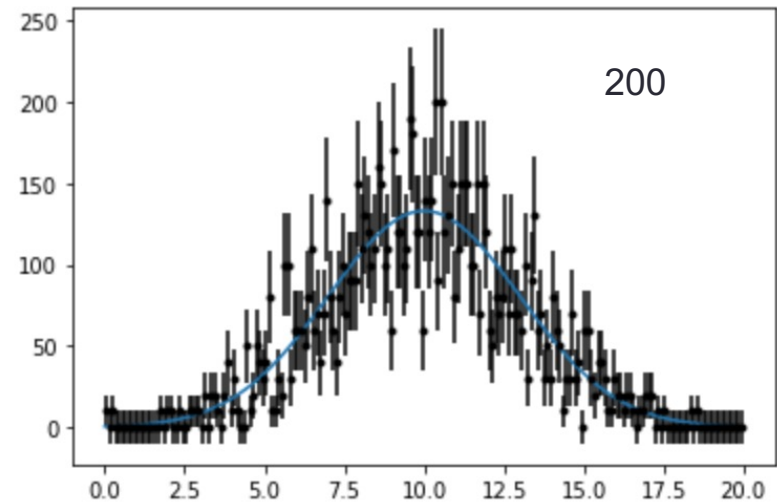
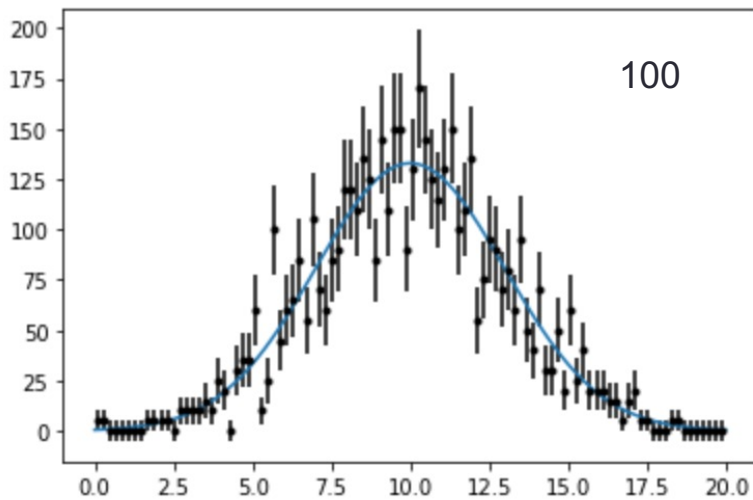
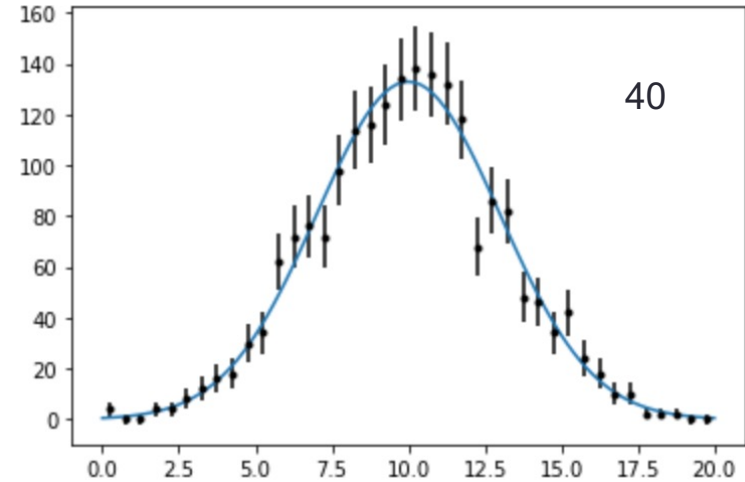
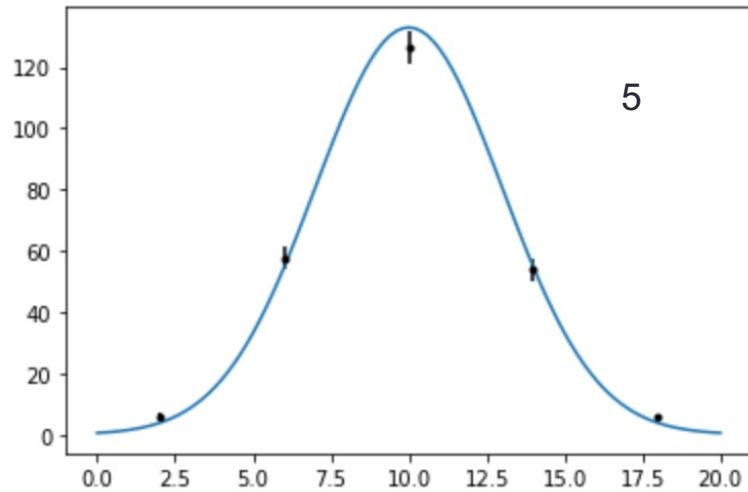
```
# Histogram them with arbitrary bin sizes
plotrange = (0.,20)
Nbins = 10
hist,bins = np.histogram(ssig,Nbins,plotrange)
hist_error = np.sqrt(hist)
hist_error = np.clip(hist_error,1.,None) # If the sqrt is 0, set it to 1.
xcent = .5*(bins[:-1]+bins[1:])
# Normalize
# Let's do it in a way that will work with arbitrary bins
binsize = bins[1:]-bins[:-1]
hist = hist/binsize
hist_error /= binsize

plt.errorbar(xcent, hist, hist_error, fmt="k.",
             label="simulation")
xlin = np.linspace(0.,20.,101)
pl=plt.plot(xlin,Nsignal*stats.norm.pdf(xlin,loc=mu,scale=sig),label='model')
plt.xlabel('x')
plt.ylabel('N/bin')
plt.legend()
plt.show()
```





# Other Binnings





# Arbitrary Distributions

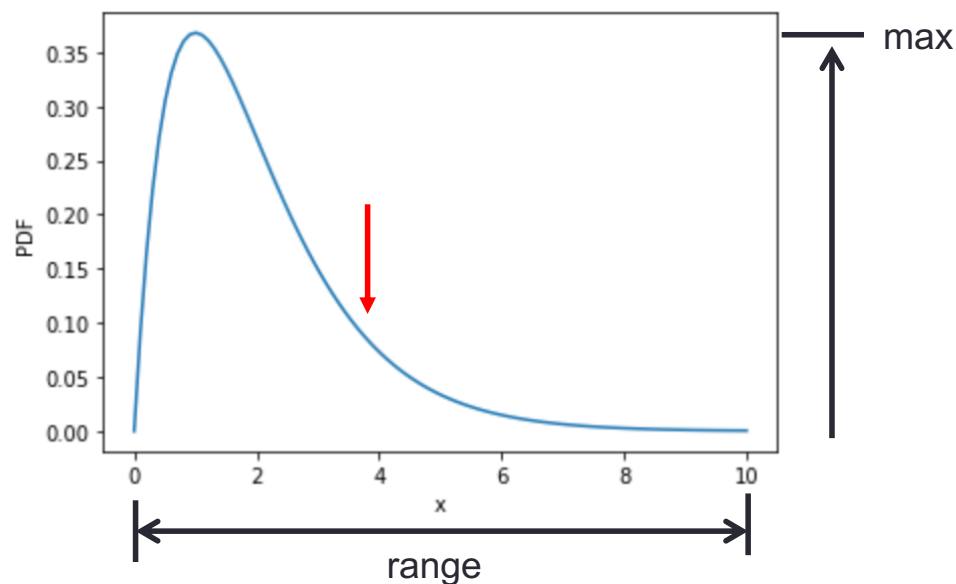
- In many cases, we want to generate a distribution based on an arbitrary parametrization.
  - Example, simulating background to a physics signal
- In this case, we can use a “Monte Carlo” technique to simulate any distribution





# Procedure

- Example:

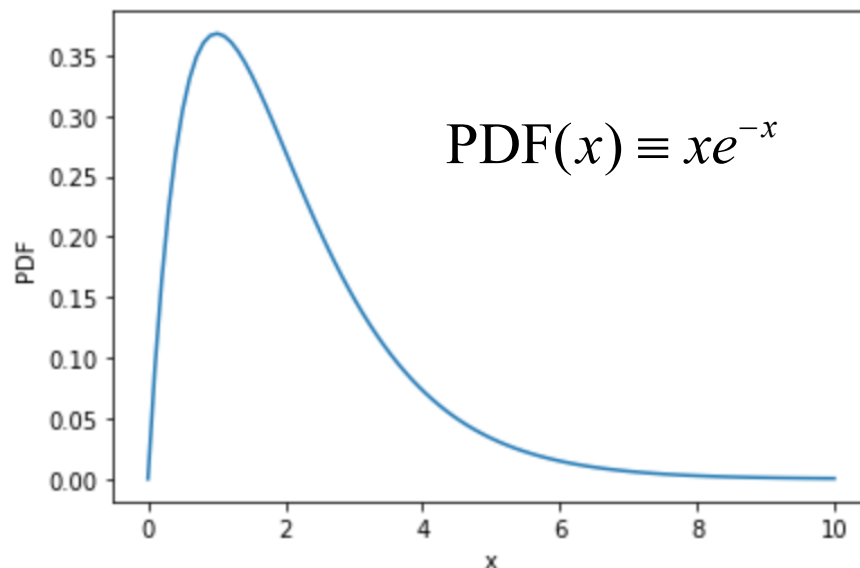


- Step 0: Choose a range for your random numbers
- Step 1: Generate a random value of  $x$  which is *uniformly* distributed over the range.
- Step 2: Evaluate the PDF at this point
- Step 3: Is the PDF value at this point greater than a random number between 0 and that maximum value of the function?
  - Yes? Return the value of  $x$
  - No? Go back to step 1



# Example

- Generate random numbers with a distribution



Already normalized.

$$\int_0^{\infty} \text{PDF} dx = \int_0^{\infty} x e^{-x} dx = 1$$

$$\frac{d}{dx} \text{PDF}(x) = (1 - x) e^{-x}$$

$$\rightarrow x_{\text{max}} = 1$$

$$(\text{PDF})_{\text{max}} = e^{-1}$$

- It looks like a range of 0 to 10 is reasonable, so let's code this up...



# Example Functions

```
[24... # Returns N 1-D random numbers based on the function myfunc, in the range xlo to xhi
# funcmax is the maximum value of funcmax
#
def mydist(func,N=1,xlo=0.,xhi=1.,funcmax=1.):
    s = np.empty(N)          # Generate an empty vector N long
    for i in range(0,N):
        while (True):        # Loop until we find a "good" number
            x = np.random.uniform(xlo,xhi)
            if(func(x)>=funcmax*np.random.uniform()):
                break         # Exit the generation loop
        s[i]=x
    return s
```

```
[25... # Test distribution function
# function x*exp(-x)
#
def distfunc(x):
    val = x*np.exp(-x)      # This is already normalized to 1
    return val
```



```
[28... # Plot a histogram of my test function
Nsignal = 10000
xlo=0.
xhi=10.
plotrange = (xlo,xhi)
Nbins = 50
ssig = mydist(distfunc,Nsignal,xlo=xlo,xhi=xhi,funcmax=np.exp(-1.))
# Now do everything the way we did it before
hist,bins = np.histogram(ssig,Nbins,plotrange)
hist_error = np.sqrt(hist)
hist_error = np.clip(hist_error,1.,None) # If the sqrt is 0, set it to 1.
binsize = bins[1]-bins[0]
binloc = bins[0:-1]+.5*binsize      # Move location to the middle of the bin
hist = hist/binsize                  # Normalize
hist_error = hist_error/binsize      # Normalize error
plt.errorbar(binloc, hist, hist_error, fmt="k.",
              label="simulation")
xlin = np.linspace(xlo,xhi,100)
pl=plt.plot(xlin,Nsignal*distfunc(xlin))
lx = plt.xlabel("x")
ly = plt.ylabel("dN/dx")
```

