RandomNumbers-FirstSteps

April 26, 2021

1 Random Numbers

In this notebook we will experiment with random numbers and probabilities. We will use some functions from SciPy, an extension of Python that provides many powerful tools to do science. To use these functions coneniently, execute the folowing cell (i.e., select it and press Shift+Return):

[1]:
```
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

When evaluating this cell, you might have gotten a message ``Populating the interactive namespace from numpy and matplotlib''. You can safely ignore it. Now we are ready to start!

1.0.1 Rolling Dice

Rolling a fair die should result in one of 6 possible outcomes (1, 2, 3, 4, 5 or 6), each with equal probability of 1/6. SciPy provides a function called ``randint'' that does something similar. You can read about ``randint'', or most other functions, simply by evaluating a cell with the function name, followed by a question mark. For example, when you evaluate the following cell, a window will pop up to tell you about the ``randint'' function (fell free to close it when you are done):

[2]:
```
randint?
```

```
[0;31mDocstring:[0m
randint(low, high=None, size=None, dtype=int)

Return random integers from "0060low"0060 (inclusive) to "0060high"0060 (exclusive).

Return random integers from the "discrete uniform" distribution of
the specified dtype in the "half-open" interval ["0060low"0060, "0060high"0060). If
"0060high"0060 is None (the default), then results are from [0, "0060low"0060).

.. note::
    New code should use the "0060"0060integers"0060"0060 method of a "0060"0060default_rng()"00
    instance instead; see "0060random-quick-start"0060.

Parameters
----------
```

```
low : int or array-like of ints
    Lowest (signed) integers to be drawn from the distribution (unless
    "0060"0060high=None"0060"0060, in which case this parameter is one above the
    *highest* such integer).
high : int or array-like of ints, optional
    If provided, one above the largest (signed) integer to be drawn
    from the distribution (see above for behavior if "0060"0060high=None"0060"0060).
    If array-like, must contain integer values
size : int or tuple of ints, optional
    Output shape.  If the given shape is, e.g., "0060"0060(m, n, k)"0060"0060, then
    "0060"0060m * n * k"0060"0060 samples are drawn.  Default is None, in which case a
    single value is returned.
dtype : dtype, optional
    Desired dtype of the result. Byteorder must be native.
    The default value is int.

    .. versionadded:: 1.11.0

Returns
-------
out : int or ndarray of ints
    "0060size"0060-shaped array of random integers from the appropriate
    distribution, or a single such random int if "0060size"0060 not provided.

See Also
--------
random_integers : similar to "0060randint"0060, only for the closed
    interval ["0060low"0060, "0060high"0060], and 1 is the lowest value if "0060high"0060 is
    omitted.
Generator.integers: which should be used for new code.

Examples
--------
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0]) # random
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1], # random
       [3, 2, 2, 0]])

Generate a 1 x 3 array with 3 different upper bounds

>>> np.random.randint(1, [3, 5, 10])
array([2, 2, 9]) # random
```

```
Generate a 1 by 3 array with 3 different lower bounds

>>> np.random.randint([1, 5, 7], 10)
array([9, 8, 7]) # random

Generate a 2 by 4 array using broadcasting with dtype of uint8

>>> np.random.randint([1, 3, 5, 7], [[10], [20]], dtype=np.uint8)
array([[ 8,  6,  9,  7], # random
       [ 1, 16,  9, 12]], dtype=uint8)
[0;31mType:[0m      builtin_function_or_method
```

From the documentation, it seems that ``randint(6)'' should result in a random integer that is greater or equal than 0 and less than 6. Let's try it:

```
[7]: randint(6)
```

```
[7]: 4
```

To emulate rolling a die we want a number that is greater or equal than 1 and less or equal than 6, which we can accomplish by simply adding one to the result of randint:

```
[ ]: randint(6)+1
```

Now let's roll the die 100 times, i.e, let's create 100 independt random numbers. We will save the result in a variable; let's call it x:

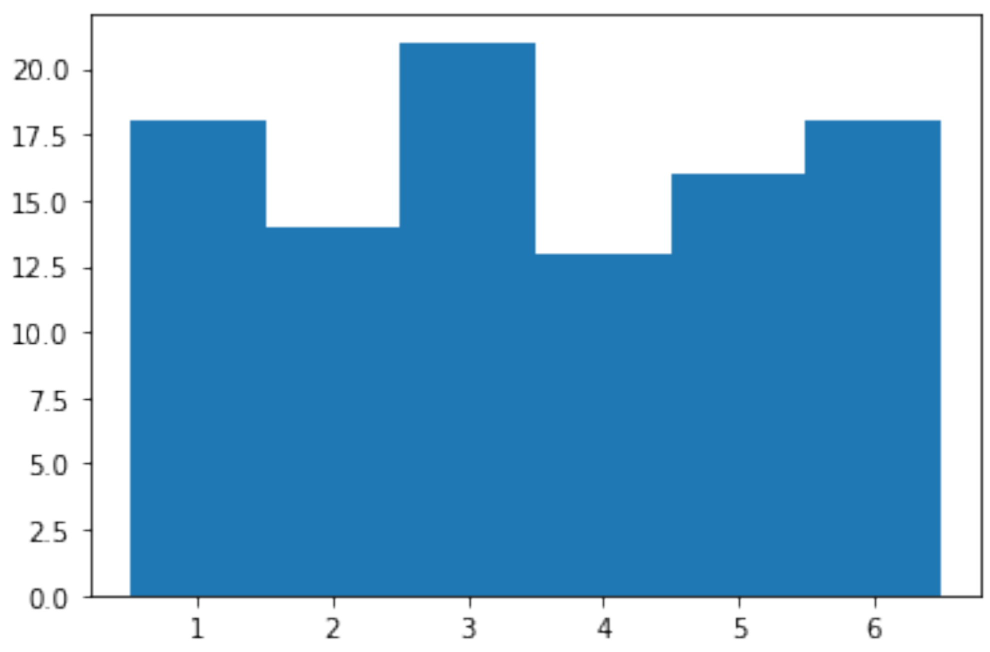
```
[8]: x=randint(6, size=100)+1
     print (x)
```

```
[6 3 6 3 2 4 2 6 1 3 1 1 5 3 2 6 1 3 1 1 4 2 3 2 1 5 3 3 6 3 3 2 3 5 4 4 3
 5 4 1 6 3 1 5 2 1 4 3 2 3 1 3 6 6 3 2 4 2 4 6 5 6 3 6 1 5 4 5 5 4 4 5 2 1
 4 1 5 1 1 5 6 5 3 5 5 6 2 6 6 2 6 6 4 5 3 2 1 6 1 3]
```

Statistically, each of the 6 possible outcomes should occur equally often. To test this we can make a histogram of the 100 observations we just made. You can use the ``hist'' function to do that quickly (remember that you can execute the command ``hist?'' to learn about the hist function):

```
[9]: hist(x,range=(0.5,6.5),bins=6)
```

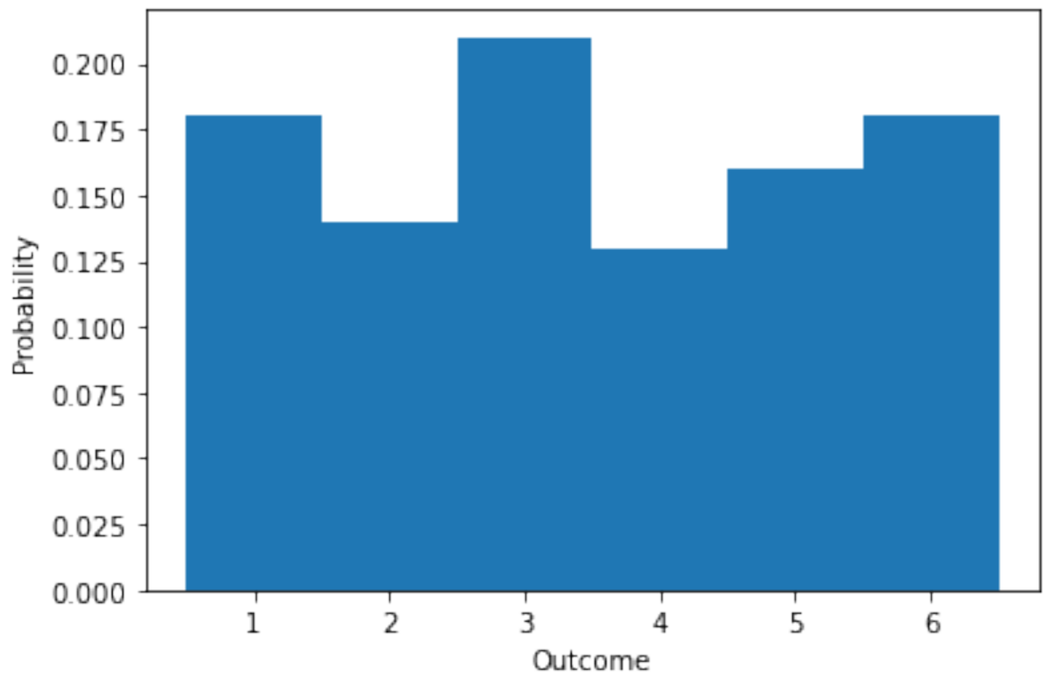
```
[9]: (array([18., 14., 21., 13., 16., 18.]),
      array([0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5]),
      <a list of 6 Patch objects>)
```



Beautiful! Notice that we created a histogram with 6 evenly spaced bins between 0.5 and 6.5. Rather than producing a histogram, we can also use the ``hist'' function to compute the probability distribution with the additional keyword ``normed=True''. While we are at it, let's also include some axis labels:

[10]:
```
hist(x,range=(0.5,6.5),bins=6,density=True)
xlabel("Outcome")
ylabel("Probability")
```

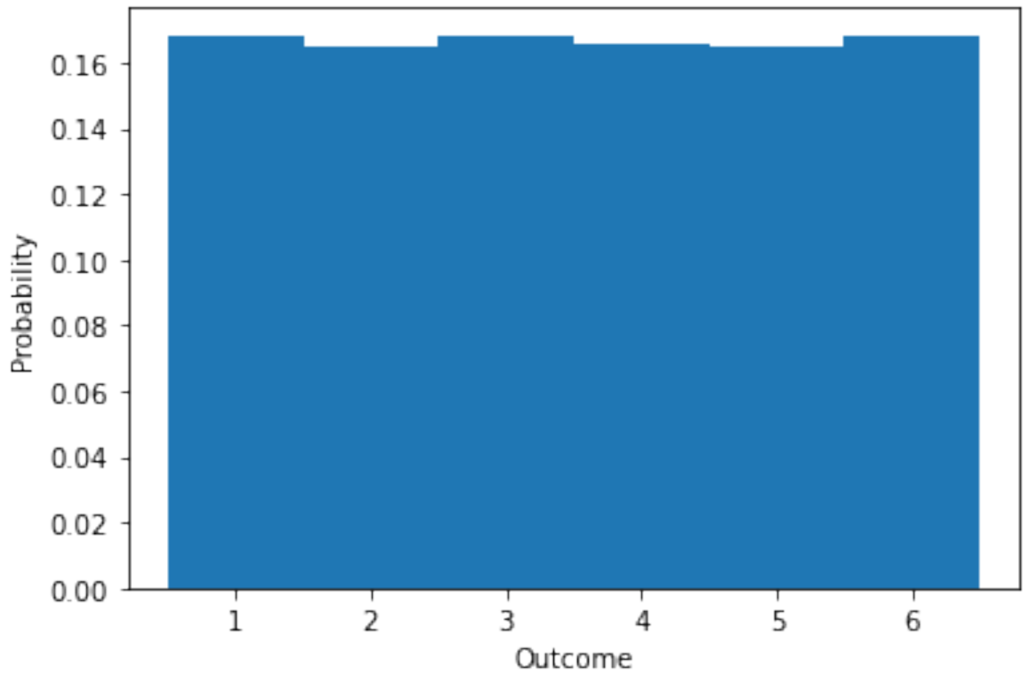
[10]:
```
Text(0, 0.5, "0027Probability"0027)
```



You see that the probability of each possible outcome is somewhere close to 1/6. Let's do this again, but this time we roll the die 100000 times (don't try to do that with a real die):

```
[11]: x=randint(6, size=100000)+1
      hist(x,range=(0.5,6.5),bins=6,density=True)
      xlabel("Outcome")
      ylabel("Probability")
```

```
[11]: Text(0, 0.5, "0027Probability"0027)
```



You see that if you have many more samples, your observation will be much closer to the true probability distribution.

We can not only draw histograms, we can also quickly compute the mean of our observations:

[12]:
```
print ("Mean:", mean(x))
print ("Variance:", var(x))
```

```
Mean: 3.49828
Variance: 2.9263570416
```

1.0.2 Continuous Random Numbers

In the die rolling example above we considered an expeirment that had only 6 possible outcomes. Now let's look at a process with a continuous sample space: picking a random number between 0 and 1 with uniform probability.

Creating such numbers can be easily done in Python; you can read about it in the documentation of the ``rand'' function:

[13]:
```
rand?
```

```
[0;31mDocstring:[0m
rand(d0, d1, ..., dn)

Random values in a given shape.
```

```
.. note::
    This is a convenience function for users porting code from Matlab,
    and wraps "0060random_sample"0060. That function takes a
    tuple to specify the size of the output, which is consistent with
    other NumPy functions like "0060numpy.zeros"0060 and "0060numpy.ones"0060.

Create an array of the given shape and populate it with
random samples from a uniform distribution
over "0060"0060[0, 1)"0060"0060.

Parameters
----------
d0, d1, ..., dn : int, optional
    The dimensions of the returned array, must be non-negative.
    If no argument is given a single Python float is returned.

Returns
-------
out : ndarray, shape "0060"0060(d0, d1, ..., dn)"0060"0060
    Random values.

See Also
--------
random

Examples
--------
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
[0;31mType:[0m      builtin_function_or_method
```

So, let's go ahead and generate 100 such random numbers:

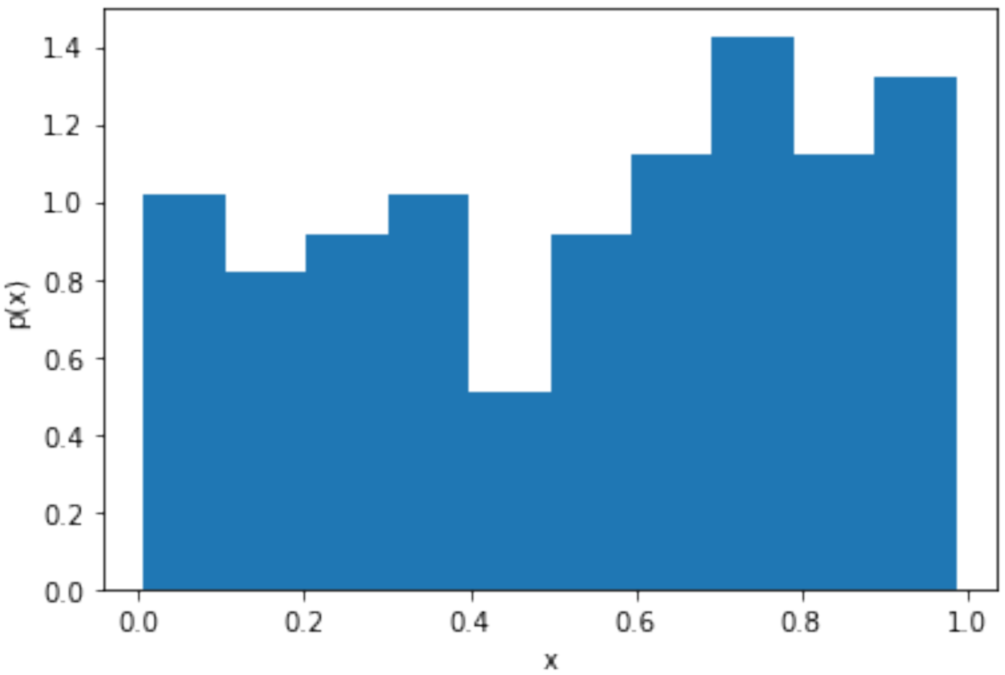
[14]:
```
x=rand(100)
print (x)
```

```
[0.68815902 0.88637705 0.55708936 0.59854085 0.42875998 0.9111312
 0.98290583 0.06866527 0.56250665 0.14641319 0.35021375 0.54879893
 0.01755661 0.76870735 0.20569875 0.37130946 0.017682   0.82924424
 0.13778337 0.16855095 0.74632052 0.61351806 0.30909803 0.8265152
 0.25035231 0.51505246 0.78048977 0.72948129 0.09805457 0.65696482
 0.68976606 0.25523254 0.98770715 0.96944071 0.37416471 0.70391803
 0.94282506 0.74082752 0.94193627 0.9847373  0.08574497 0.25585229
 0.16843829 0.92967006 0.22109882 0.12322559 0.87856505 0.15698083
 0.31094478 0.46694811 0.80377383 0.25551756 0.70772748 0.68579701
 0.86068254 0.02930845 0.64895257 0.78978701 0.66761535 0.62799929
 0.73295794 0.19343026 0.79138302 0.9871116  0.00636273 0.95728485
 0.74624634 0.88149768 0.92085504 0.03363315 0.2054777  0.86076398
 0.82275925 0.9124652  0.07624698 0.31084641 0.21519738 0.48080884
 0.37112408 0.59495376 0.75480938 0.44378643 0.36938373 0.34273565
 0.67495658 0.43382423 0.06570394 0.74414588 0.55817903 0.58099578
 0.88197237 0.16664838 0.95671045 0.32797173 0.79431229 0.66877857
 0.28050916 0.58540629 0.77114526 0.54081696]
```

Here is the probability distribution we obtain from this dataset, together with the mean and the variance:

[15]:
```
hist(x,density=True)
xlabel("x")
ylabel("p(x)")
print ("Mean:", mean(x))
print ("Variance:", var(x))
```

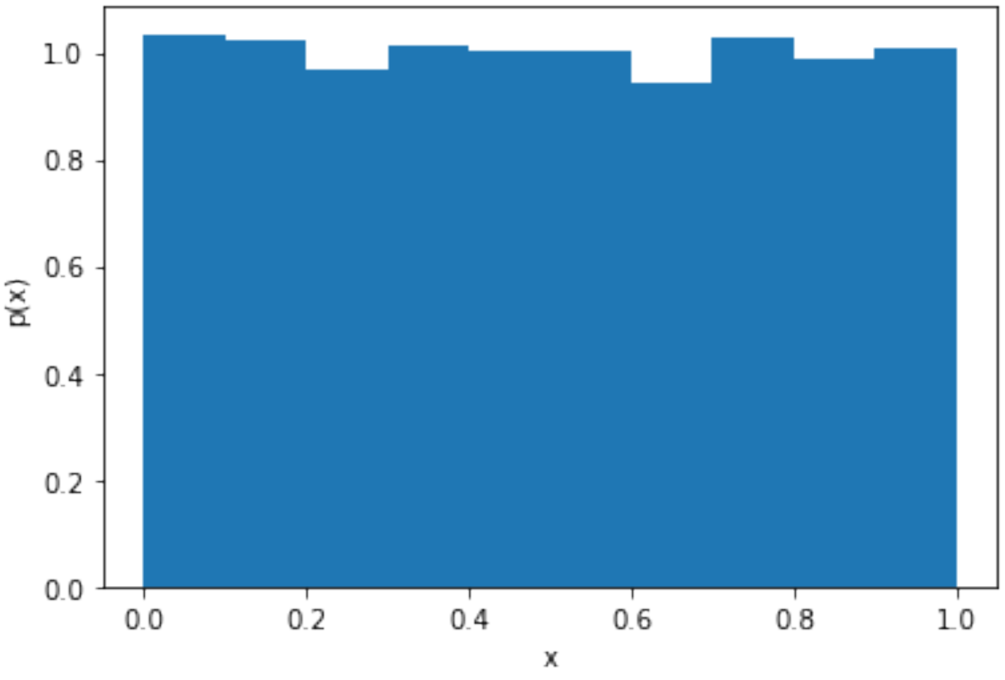
```
Mean: 0.5408132437222097
Variance: 0.08918042369622439
```



Let's see how those numbers change when we use a larger sample size, for example 10000 samples:

[16]:
```
x=rand(10000)
hist(x,density=True)
xlabel("x")
ylabel("p(x)")
print ("Mean:", mean(x))
print ("Variance:", var(x))
```

```
Mean: 0.4983180161870467
Variance: 0.08393729915318966
```



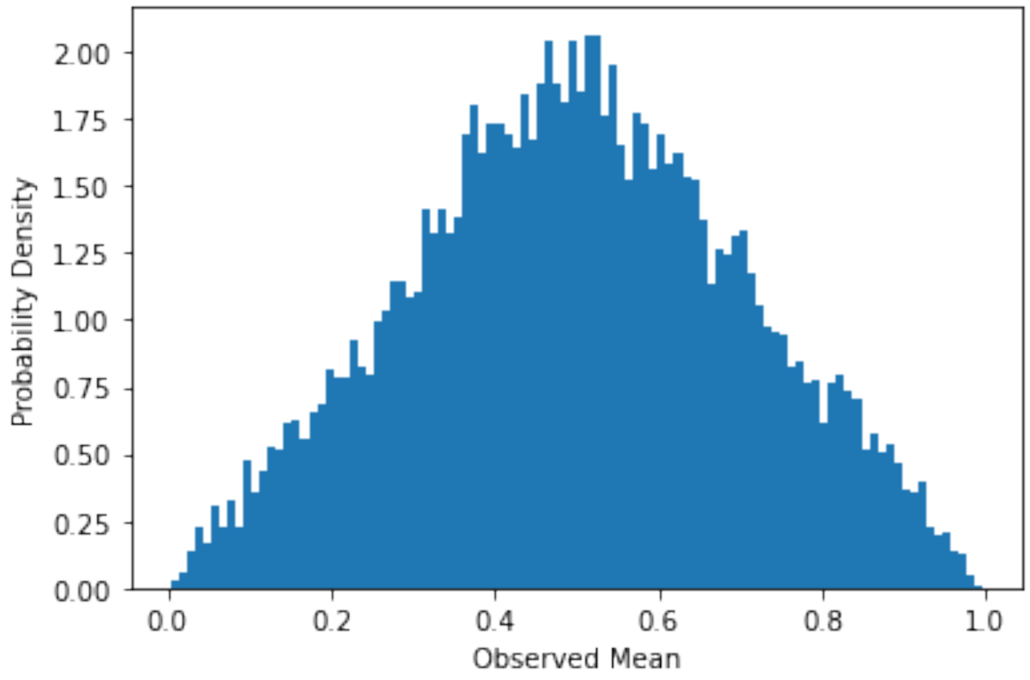
So far we have looked at the probability distribution of picking a single number randomly between 0 and 1. Now we ask a different question: What is the probability distribution of the *average* of picking N such numbers?

We will pick N random numbers, calculate the average, and do this over and over again until we have enough oberservations of this average to make a histogram. We will start with an empty list (called ``averages''), and add to that list each observation of the average. Let's start with a small number of N, let's say N=2:

[17]:
```
N=2
averages = []                     # this creates an empty list with the name␣
↪"averages"
for i in range(10000):            # let"0027s do the following 100000 times:
    x=rand(N)                     #     pick N independent random numbers from a␣
↪uniform distribution over [0,1]
```

```
    averages.append (mean(x))  #     calculate the average of those N numbers,␣
 ↪and add it to the list
hist(averages,bins=100,density=True)   # compute and display the histogram␣
 ↪distribution
xlabel("Observed Mean")
ylabel("Probability Density")
print ("Mean:", mean(averages))
print ("Variance:", var(averages))
```

```
Mean: 0.5004026490634754
Variance: 0.04098892701634022
```



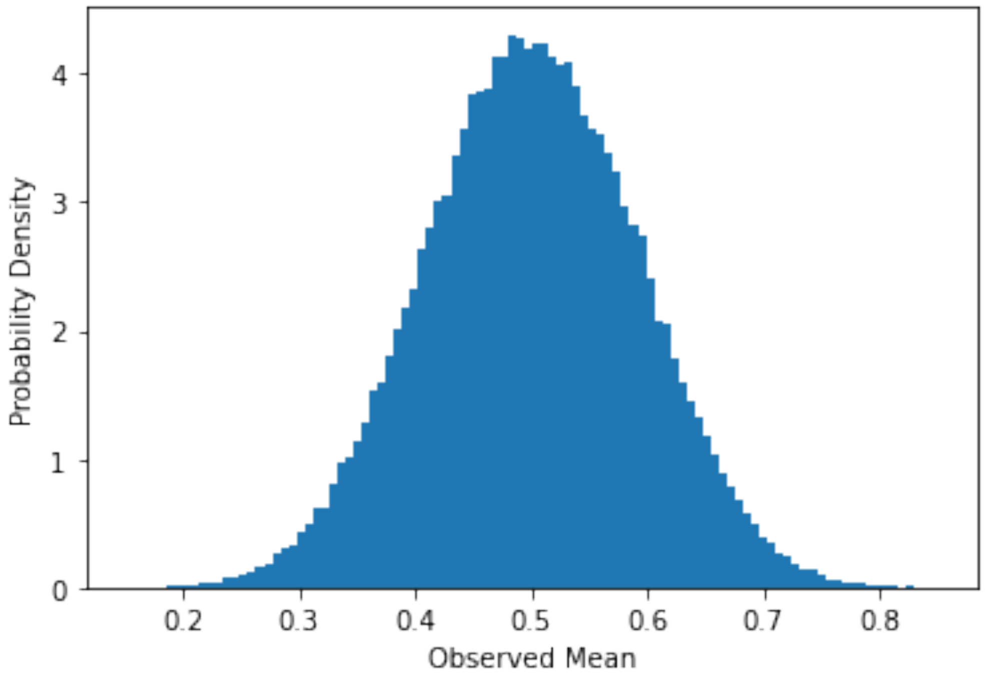
You see that the probability distribution for the average of 2 random numbers is already very different from the distribution of a single random number.

Let us do this again, but this time we average over 10 random numbers:

[18]:
```
N=10
averages = []                   # this creates an empty list with the name␣
 ↪"averages"
for i in range(100000):         # let"0027s do the following 100000 times:
    x=rand(N)                   #     pick N independent random numbers from a␣
 ↪uniform distribution over [0,1]
    averages.append (mean(x))  #     calculate the average of those N numbers,␣
 ↪and add it to the list
```

```
hist(averages,bins=100,density=True)    # compute and display the histogram␣
 ↪distribution
xlabel("Observed Mean")
ylabel("Probability Density")
print ("Mean:", mean(averages))
print ("Variance:", var(averages))
```

```
Mean: 0.5002356517926007
Variance: 0.008398043918019881
```

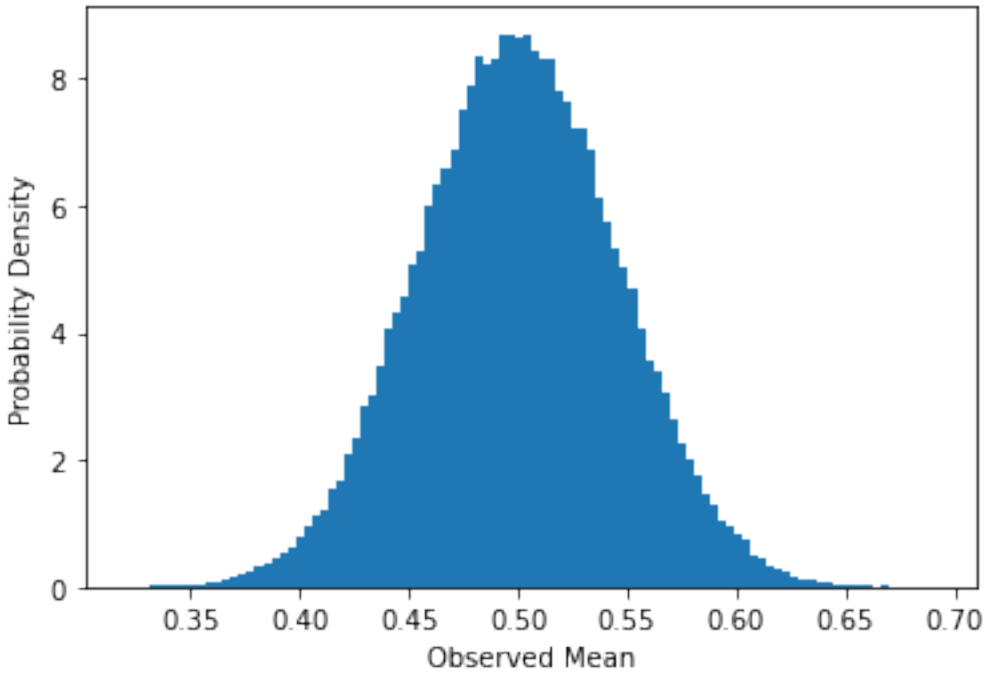


This already looks very Gaussian! Let's do this again, but this time with N=40. Before you evaluate this cell, please stop and think about how would expect the distribution to look like, and what you think the mean and variance will be based on what you've just seen!

[19]:
```
N=40
averages = []                      # this creates an empty list with the name␣
 ↪"averages"
for i in range(100000):            # let"0027s do the following 100000 times:
    x=rand(N)                      #     pick N independent random numbers from a␣
 ↪uniform distribution over [0,1]
    averages.append (mean(x))      #     calculate the average of those N numbers,␣
 ↪and add it to the list
hist(averages,bins=100,density=True)    # compute and display the histogram␣
 ↪distribution
xlabel("Observed Mean")
```

```
ylabel("Probability Density")
print ("Mean:", mean(averages))
print ("Variance:", var(averages))
```

```
Mean: 0.5000340114280194
Variance: 0.002093290537696063
```



The distribution is even more Gaussian, but it is now narrower. The mean remains the same, but the variance should have decreased by a factor of 4 -- can you guess why?

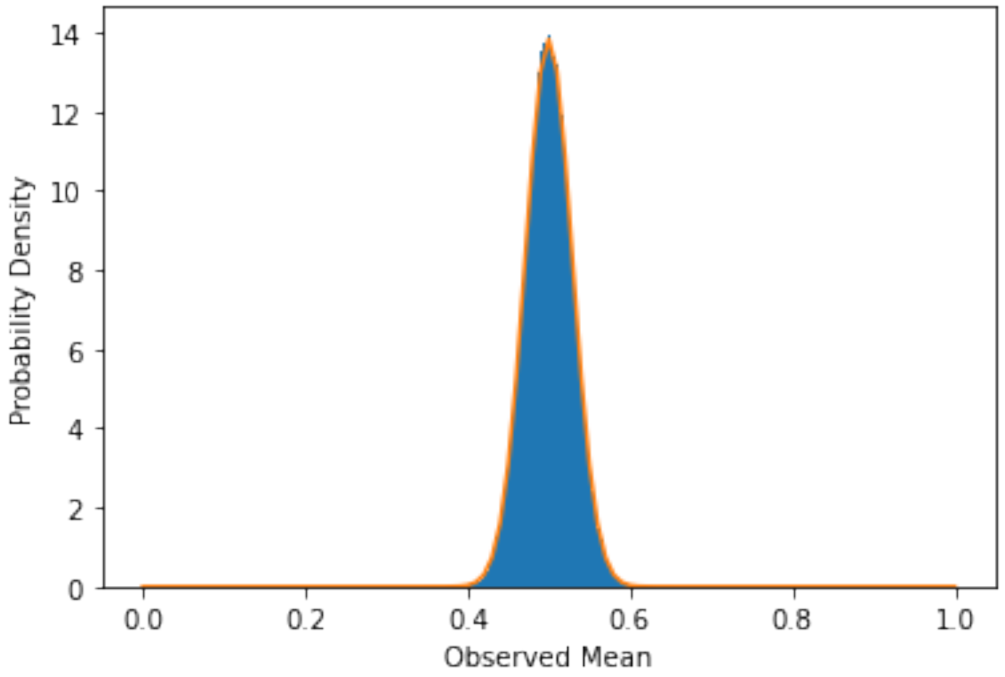
[22]:
```
def gaussian(x, N, mean, variance):
    coeff = 1/np.sqrt(2*np.pi*variance/N)
    exponential = -(x-mean)**2/(2*variance/N)
    return coeff*np.exp(exponential)
```

[28]:
```
x_range = np.linspace(0,1,101)
gaussian_x = gaussian(x_range, 100, 0.5, (1/12))
```

[30]:
```
N=100
averages = []                   # this creates an empty list with the name␣
 ↪"averages"
for i in range(100000):         # let"0027s do the following 100000 times:
    x=rand(N)                   #     pick N independent random numbers from a␣
 ↪uniform distribution over [0,1]
```

```
    averages.append (mean(x))   #     calculate the average of those N numbers,␣
 ↪and add it to the list
hist(averages,bins=100,density=True)    # compute and display the histogram␣
 ↪distribution
plot(x_range, gaussian_x)
xlabel("Observed Mean")
ylabel("Probability Density")
print ("Mean:", mean(averages))
print ("Variance:", var(averages))
```

```
Mean: 0.4999613444120334
Variance: 0.0008349441814745921
```



```
[ ]:
```