

The Random package, Defining Custom Functions

Chris Cornwell

Feb 4, 2025

Outline

Random functions in NumPy

Defining custom functions

Outline

Random functions in NumPy

Defining custom functions

The random submodule in NumPy

From documentation: Within NumPy is the submodule `random` which “implments pseudo-random number generators...with the ability to draw samples from a variety of probability distributions.”

1. Uniform discrete: to sample `m` numbers from the uniform distribution on the integers in `range(n)`, use the function `np.random.randint(n, size=m)`.
 - ▶ An optional additional integer in the arguments: give a lower bound.¹
 - ▶ If size argument not given, just one number returned. If the `size=m` is given, returns a NumPy array.

```
1 | # 20 integers, each equally likely, between 1 and 10  
2 | np.random.randint(1, 11, size=20)
```

Commands above will sample with replacement.

- ▶ To sample without replacement, use argument `replace=False`.

¹The default lower end is 0, to sample between 0 and `n-1`.

The random submodule in NumPy

2. Uniform continuous: to sample m floats in the interval $[a, b)$, use the function `np.random.uniform(a, b, size=m)`.
 - ▶ The defaults for the left & right endpoints are 0 and 1. This means that `np.random.uniform(size=m)` will sample from the unit interval.
 - ▶ An alternative: `np.random.random(size=m)`. Only samples from the unit interval, but, can get samples from $[a, b)$ by multiplying and adding (below).

```
| (b-a)*np.random.random(size=m) + a
```

The random submodule in NumPy

2. Uniform continuous: to sample m floats in the interval $[a, b)$, use the function `np.random.uniform(a, b, size=m)`.
3. Normal distribution: to sample m floats from the normal distribution, with mean μ and standard deviation σ , use the function `np.random.normal(mu, sigma, size=m)`
 - ▶ The two arguments for the mean and standard deviation have default 0 and 1, respectively.
 - ▶ Not uncommon to use the *keywords* for these arguments,² as in
`np.random.normal(loc=mu, scale=sigma, size=m)`

Larger the sample, the closer the sample distribution should be to the theoretical distribution. Can we display this in a plot?

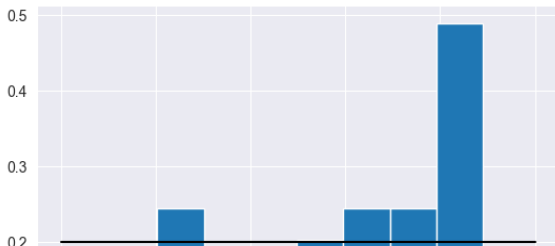
²Using keyword makes the argument not *positional* anymore; arguments coming after it must have keyword too.

Visualizing the distribution

Say that we are using the uniform distribution on the interval $[a, b)$. For a small subinterval of width δ , we have probability $\delta \frac{1}{b-a}$ of getting a sample in that interval.

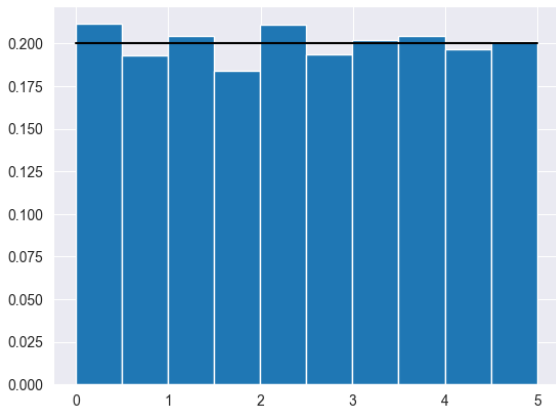
Consequence: if plot (normalized) histogram of sample over the interval $[a, b]$, being close to the distribution means bars are close to height $1/(b-a)$, on average. Below is a histogram from a sample of size 50, from `uniform(0, 5)`.

```
1 | xx = np.linspace(0,5)
2 | sample = np.random.uniform(0, 5, size=50)
3 | # will also plot horizontal line at height 1/(b-a)
4 | plt.plot(xx, [1/5]*len(xx), color='black')
5 | # "density=True" below -> divide height of each bar by number of samples
6 | plt.hist(sample, bins=10, density=True)
7 | plt.show()
```



Visualizing the distribution

A larger sample size will (on average) give a sample distribution that is closer to the *true* one. For example, by changing our previous code so that the sample size is 5000, we can see the sample distribution get much closer.



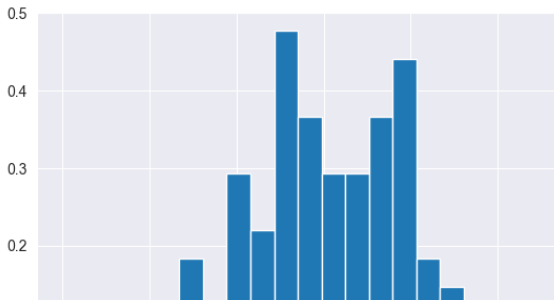
Visualizing the normal distribution

If using the normal distribution, with mean μ and st.deviation σ , the density function is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Consequence: a histogram of a sample being close to the distribution means top of bars are close to graph of $p(x)$. To make the histogram, can do following.³

```
1 | xx = np.linspace(-3,3)
2 | sample = np.random.normal(0, 1, size=100)
3 | plt.hist(sample, bins=20, density=True)
4 | plt.show()
```



Visualizing the normal distribution

Again, about the normal distribution

Some miscellany about distributions in `numpy.random`

- ▶ Many other probability distributions are implemented in `numpy.random`. See [this link](#) for information about them.
- ▶ There is a useful command to shuffle an array: randomizing the order of the entries in the array:
`np.random.shuffle(the_array)`.
 - ▶ This changes `the_array` in place.
- ▶ Not only a 1d array; can get a matrix (or any order tensor) with entries sampled from the distribution by adjusting the `size` argument.

```
1 | # to get 30x2 matrix, entries from normal N(0,1)  
2 | np.random.normal(size=(30,2))
```

Outline

Random functions in NumPy

Defining custom functions

How to define a custom function

The basic structure for a definition of a custom function has four parts: the function name, a list of arguments it takes, the “body” of the function (what happens when it is called), and what it returns.

Like this:

```
1 | def function_name(argument1, argument2):  
2 |     # in this part is the body of the function  
3 |     return function_output  
4 |
```

Example. Here is a function that takes a 1d array as input. It makes a copy of it, but in reversed order. Then it multiplies these together entry-wise, and any negative numbers obtained are replaced with 0. It then returns the resulting array of non-negative numbers.

```
1 | def my_function(v):  
2 |     reverse_v = v[::-1]  
3 |     multiplied = reverse_v * v  
4 |     zeros = np.zeros(len(v))  
5 |     return np.maximum(multiplied, zeros)
```

A custom function to help with a plot

Say that I have 500 points in the plane. Maybe they are selected randomly in some way, such as

```
p = np.random.uniform(-1, 1, size=(500,2))
```

Now, say that I want to plot them, coloring them dark blue if their dot product with the vector $(2, -2/3)$ is negative, and coloring them a *salmon* color otherwise.

An efficient way to make the array that contains the colors, ordered as the points are?

```
1 def coloring(array_of_points):
2     ref_vector = np.array([2,-2/3])
3     dotvalues = array_of_points@ref_vector
4     colors = np.array(['darkblue' if v < 0 else 'salmon' for v in dotvalues])
5     return colors
```

Additional arguments, default values, keyword arguments

```
1 def my_function(v, *, translate=2):  
2     reverse_v = v[::-1]  
3     multiplied = reverse_v * v + translate  
4     zeros = np.zeros(len(v))  
5     return np.maximum(multiplied, zeros)
```