The Random package, Defining Custom Functions

Chris Cornwell

Feb 4, 2025

Outline

Outline

Random functions in NumPy

Defining custom functions

Importing data

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.

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

   Commands above will sample with replacement.
   - To sample without replacement, use argument `replace=False`.

[1] The 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 `mu` and standard deviation `sigma`, 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)`

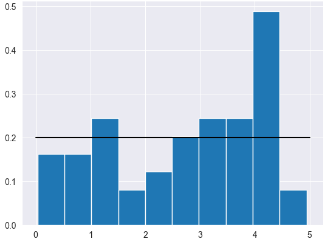
Larger the sample, the closer the distribution of that sample (i.e., normalized histogram) should be to the theoretical pdf. Can we display this in a plot?

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

Visualizing the distribution

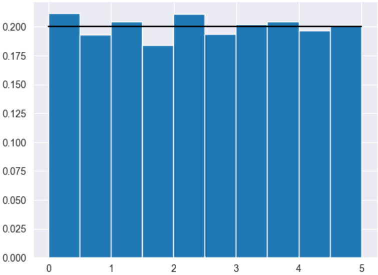
Using uniform distribution on interval $[a, b)$: for subinterval of width δ, probability $\delta \frac{1}{b-a}$ of getting a sample in that interval.
Consequence: plot (normalized) histogram of sample over $[a, b]$; *close* to the distribution means bars are close to height $1/(b-a)$.
Below: sample of size 50, from `uniform(0, 5)`.

```
xx = np.linspace(0,5)
sample = np.random.uniform(0, 5, size=50)
# will also plot horizontal line at height 1/(b-a)
plt.plot(xx, [1/5]*len(xx), color='black')
plt.hist(sample, bins=10, density=True)
plt.show()
```



Visualizing the distribution

A larger sample size will (on average) give a 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.

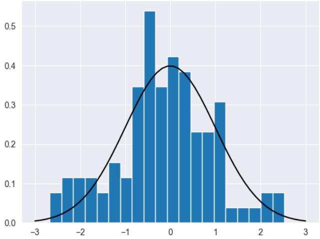


Visualizing the normal distribution

Using the normal distribution, mean μ and std.deviation σ, the density function is

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

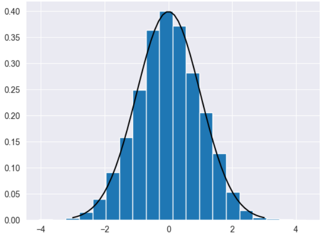
Below is a normalized histogram of a sample of size 100.[3]



[3] Doubled the sample size and the number of bins, due to greater variability in the pdf of the distribution.

Visualizing the normal distribution

A larger sample size gives a distribution that is closer to the bell curve.



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.

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

Outline

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:

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

```

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.

```
def my_function(v):
    reverse_v = v[::-1]
    multiplied = reverse_v * v
    zeros = np.zeros(len(v))
    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?

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

Additional arguments, default values, keyword arguments

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

Outline

Random functions in NumPy

Defining custom functions

Importing data

Importing data with Pandas

For this first example, we'll keep working with data very simple.
Assumption: data is contained in a CSV file, representing a "spreadsheet" of values in columns.

1. Import Pandas, command: `import pandas as pd`.
2. If file is in same folder as your Jupyter notebook, and called `file1.csv`, then assign

```
df = pd.read_csv('file1.csv')
```

The output, `df`, is a Pandas DataFrame. Assuming your CSV file had a first line with column headings ('column1', 'column2',...etc.), the values in the first column, type `df['column1']`.
A column of a dataframe has a method, .to_numpy(), that converts it to a NumPy array.

Regression line on data in plane

For the data set that was provided, assign it as a dataframe, like with variable df on last slide.
Make two NumPy arrays called and from columns ’column1’ and ’column2’. These will be x-coordinates and y-coordinates. To plot the points in the plane, use the scatter function in `pyplot`.

```
plt.scatter(x, y)
plt.show()
```

Getting the “best fit” regression line[4] is straightforward in NumPy.

```
# returns slope and intercept of best fit line
np.polyfit(x, y, 1)
```

[4] Considering x as the independent variable and y as the *response* variable.