The Random package, Defining Custom Functions

Chris Cornwell

Feb 4, 2025

Outline

Random functions in NumPy

Defining custom functions

Broadcasting and efficient operations

Outline

The `random` module in NumPy

From documentation: Within NumPy is the package `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` module 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` module 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 sample distribution should be to the theoretical distribution. Can we display this in a plot?

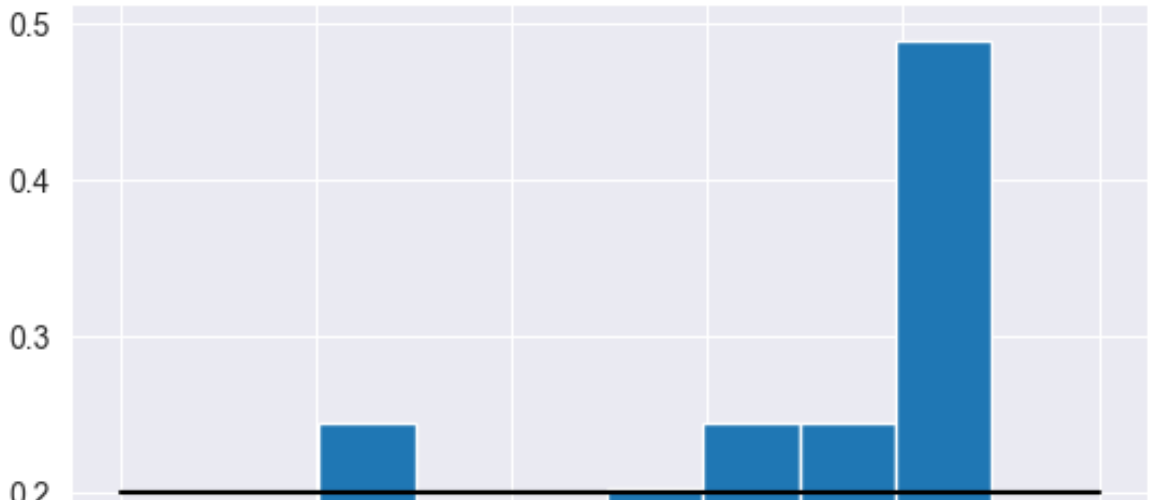
[2] Using keyword makes the argument not *positional* anymore; arguments coming after it must have keyword too.

Visualizing the distribution

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.

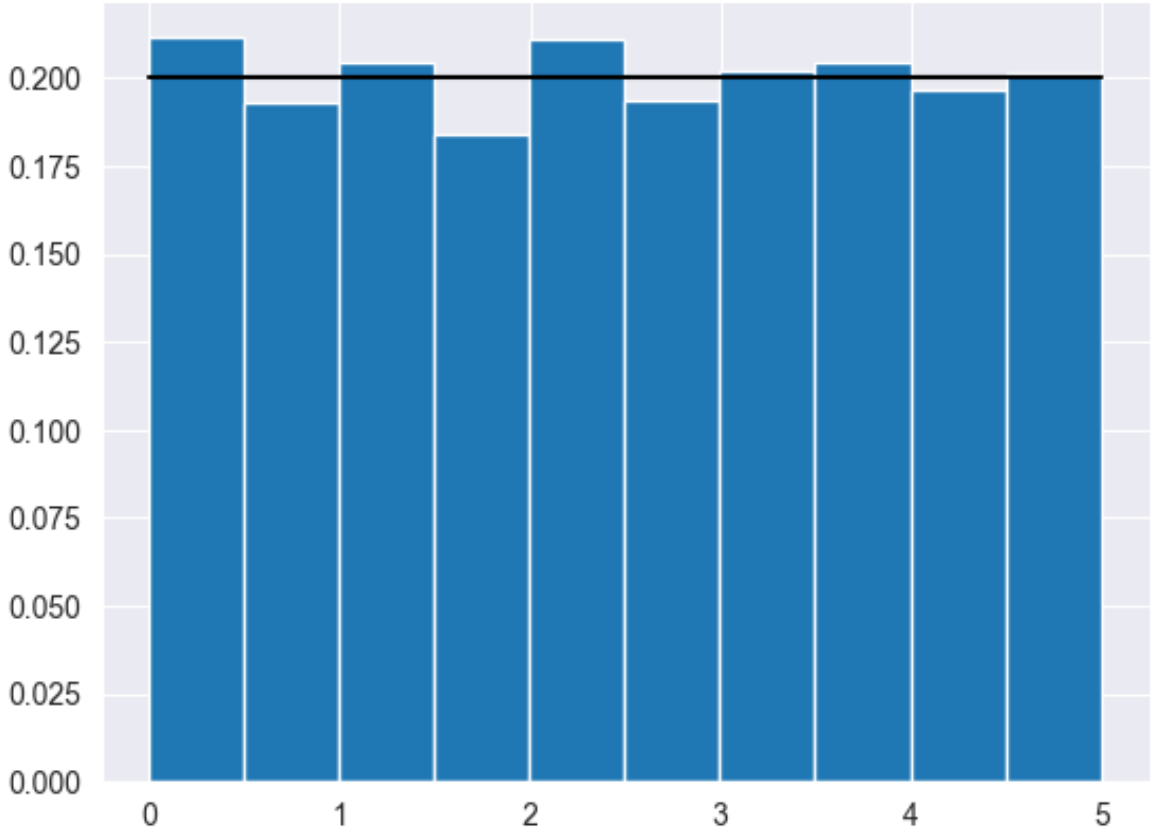
Consequence: if plot (normalized) histogram of sample over the interval $[a, b]$, being close to the distribution means bars are close to height $1/(b-a)$, on average. Below is a histogram from a sample of size 50, from `uniform(0, 5)`.

```
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')
# "density=True" below -> divide height of each bar by number of samples
plt.hist(sample, bins=10, density=True)
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.



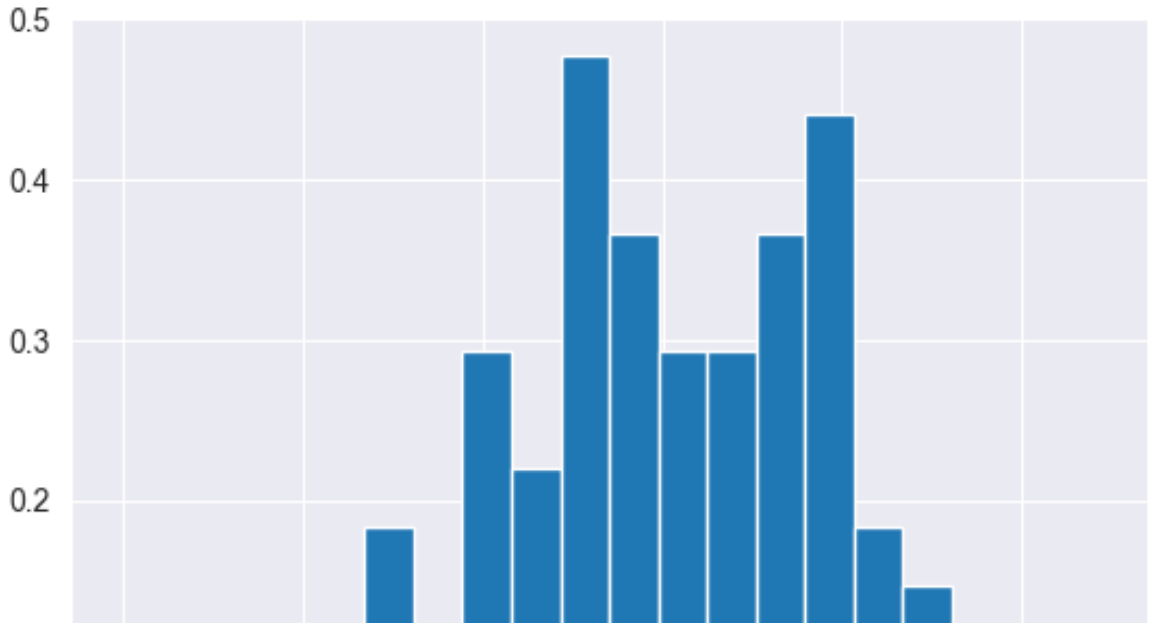
Visualizing the normal distribution

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

Outline

Constructing special matrices

Some types of matrices are used a lot; would be cumbersome to always write the row lists ourselves (e.g., in a 100 × 100 matrix).

Zero matrix: The command `np.zeros((m, n))` constructs an $m \times n$ matrix with all entries equal to zero.

Diagonal matrix: If `d` is a 1d array of length n, the command `np.diag(d)` constructs an $n \times n$ diagonal matrix which has `d` as its diagonal entries.

Identity matrix: The command `np.eye(n)` constructs the $n \times n$ identity matrix.

Extracting part of matrix: May want to get part of a matrix. To get a submatrix from consecutive rows and columns, use slicing. Also, here are functions that return part of the matrix (other entries being set to `0`).

```
# return lower triangular part (at or below the diagonal)
np.tril(A)
# return upper triangular part (at or above the diagonal)
np.triu(A)
# return the diagonal of A
np.diag(A)
```

Using NumPy for linear algebra

In addition to the product operations on arrays, NumPy has a library (`linalg`) with many functions for linear algebra.
Examples:

1. If M is a square matrix, can compute $\det(M)$ with the command `np.linalg.det(M)`.
2. When M is invertible, can compute M^{-1} with the command `np.linalg.inv(M)`.
3. If M is a square matrix, can compute eigenvalues and eigenvectors with `np.linalg.eig(M)`.

There are many other linear algebra functions. Some are only implemented for square matrices (and perhaps only invertible ones), even though it would make sense to have them work more generally – for example, `np.solve(A, b)` only solves the system $A\mathbf{x} = b$ if `A` is a square invertible matrix.

Solving a linear system & Errors

To solve $A\mathbf{x} = \mathbf{b}$, with a square invertible matrix `A` and vector `b` of the right size, you can use `np.linalg.solve(A, b)`.
What happens when `A` is not square? Execute the following code in Python.

```
A = np.array([[1, 2, 3], [1, 4, -1]])
b = np.array([1, -5])
# system has solution x = [0, -1, 1]
# but next line raises an error
x = np.linalg.solve(A, b)
```

A message is generated about the error. It gives you helpful information, if it can. In this case, it is a `LinAlgError` with the message `Last 2 dimensions of the array must be square`.
Spend time trying to use error messages to understand issues in your code. Also, have healthy skepticism about AI assistants. They hallucinate; error messages don't.[4]

[4] While writing this slide, Github Copilot suggested I write that it would be a `ValueError` from *mismatched dimensions*: rows of `A` being size `(3,)` and the vector `b` being size `(2,)`.

Outline

Broadcasting, universal functions

Say that you have a 1d array and you want to make array with square root the entries.

First thought: use a loop, taking square root (and assigning) as you go through items in the array.

NumPy has an efficient way to handle it, called *broadcasting*. If `v` is your array, then you can simply type

```
sqrt_v = np.sqrt(v)
```

The function `np.sqrt()` takes the square root of each entry in `v`; you don't need to write the for loop.[5]

Functions that work on arrays this way are quite common in NumPy. They are called **ufuncs** (universal functions).
Other examples of ufuncs in NumPy:

`np.abs()`, `np.sum()`, `np.maximum()`, `np.minimum()`, `np.exp()`, `np.log()`.

[5] Technically, there's a for loop in the background, but it happens in C and works much faster.

More on broadcasting

Many basic operations with NumPy arrays use broadcasting. Here are a few examples with an array `v`.

1. To add the same scalar, say 3, to every array entry: type `v+3`.
2. To multiply every entry by 3: type `3*v`.
3. To square every entry of an array: type `v**2`.
4. To multiply `v` by another array `w`, entry-wise[6]: type `v*w`.

Everything mentioned here works just as well on matrices (2d arrays), and generally on any `nd` array (higher order tensors).

Exercise.
Write out code that uses broadcasting to create a 100×100 matrix where all non-diagonal entries are -1 and all diagonal entries are 2.

[6] In mathematics, this product on vectors is called the Hadamard product.

Experiment with runtime for universal function

To check the efficiency of broadcasting, use the `time` package.
Beforehand, make sure that you imported both `numpy` and `time` (see slide in first section).
Something simple: from a large identity matrix, we will get the exponential of the matrix (apply the function e^x to every entry).
First, we use a `for` loop. Run the code below in your Jupyter notebook.

```
id_matrix = np.eye(1000)
exp_matrix = np.zeros((1000, 1000))
start = time.time()
for i in range(1000):
    for j in range(1000):
        exp_matrix[i,j] = np.exp(id_matrix[i,j])
end = time.time()
print(f"Seconds taken: {end-start}.")
```

The output gives the number of seconds to run the computation.
The exact time will vary based on your computer. Mine took around 0.55 seconds.

Experiment with runtime for universal function

Now, we will use broadcasting to compute the exponential of the identity matrix.

Run the following code in your Jupyter notebook.

```
id_matrix = np.eye(1000)
exp_matrix = np.zeros((1000, 1000))
start = time.time()
exp_matrix = np.exp(id_matrix)
end = time.time()
print(f"Seconds taken: {end-start}.")
```

Again, the output is the number of seconds of runtime. For this approach with `np.exp()`, my computer took around 0.0045 seconds. That is over 100 times faster than writing the loop!