Random

Generate numbers and arrays from different random distributions.

Chapter Goals:

- Learn about random operations in NumPy
- Write code using the `np.random` submodule

A. Random integers

Similar to the Python `random` module, NumPy has its own submodule for pseudo-random number generation called `np.random`. It provides all the necessary randomized operations and extends it to multi-dimensional arrays. To generate pseudo-random integers, we use the `np.random.randint (https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.randint.html#numpy.random.randint)` function.

The code below shows example usages of `np.random.randint`.

```
1   print(np.random.randint(5))
2   print(np.random.randint(5))
3   print(np.random.randint(5, high=6))
4
5   random_arr = np.random.randint(-3, high=14,
6                                  size=(2, 2))
7   print(repr(random_arr))
```

▷   💾 ↩ ⌐⌐

✕

Output   0.543s

```
0
0
5
array([[12,  6],
       [11,  2]])
```

The `np.random.randint` function takes in a single required argument, which actually depends on the `high` keyword argument. If `high=None` (which is the default value), then the required argument represents the upper (exclusive) end of the range, with the lower end being 0. Specifically, if the required argument is n, then the random integer is chosen uniformly from the range $[0, n)$.

If `high` is not `None`, then the required argument will represent the lower (inclusive) end of the range, while `high` represents the upper (exclusive) end.

The `size` keyword argument specifies the size of the output array, where each integer in the array is randomly drawn from the specified range. As a default, `np.random.randint` returns a single integer.

B. Utility functions

Some fundamental utility functions from the `np.random` module are `np.random.seed` (https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.seed.html#numpy.random.seed) and `np.random.shuffle` (https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.shuffle.html#numpy.random.shuffle). We use the `np.random.seed` function to set the random seed (https://en.wikipedia.org/wiki/Random_seed), which allows us to control the outputs of the pseudo-random functions. The function takes in a single integer as an argument, representing the random seed.

The code below uses `np.random.seed` with the same random seed. Note how the outputs of the random functions in each subsequent run are identical when we set the same random seed.

```
1  np.random.seed(1)
2  print(np.random.randint(10))
3  random_arr = np.random.randint(3, high=100,
4                                 size=(2, 2))
5  print(repr(random_arr))
6
7  # New seed
```

```
 8  np.random.seed(2)
 9  print(np.random.randint(10))
10  random_arr = np.random.randint(3, high=100,
11                                  size=(2, 2))
12  print(repr(random_arr))
13
14  # Original seed
15  np.random.seed(1)
16  print(np.random.randint(10))
17  random_arr = np.random.randint(3, high=100,
18                                  size=(2, 2))
19  print(repr(random_arr))
```

Output   0.622s

```
5
array([[15, 75],
       [12, 78]])
8
array([[18, 75],
       [25, 46]])
5
array([[15, 75],
       [12, 78]])
```

The `np.random.shuffle` function allows us to randomly shuffle an array. Note that the shuffling happens in place (i.e. no return value), and shuffling multi-dimensional arrays only shuffles the first dimension.

The code below shows example usages of `np.random.shuffle`. Note that only the rows of `matrix` are shuffled (i.e. shuffling along first dimension only).

```
1  vec = np.array([1, 2, 3, 4, 5])
2  np.random.shuffle(vec)
3  print(repr(vec))
4  np.random.shuffle(vec)
5  print(repr(vec))
6
```

```
 7  matrix = np.array([[1, 2, 3],
 8                     [4, 5, 6],
 9                     [7, 8, 9]])
10  np.random.shuffle(matrix)
11  print(repr(matrix))
```

Output   0.696s

```
array([1, 3, 4, 2, 5])
array([5, 2, 3, 1, 4])
array([[4, 5, 6],
       [1, 2, 3],
       [7, 8, 9]])
```

C. Distributions

Using `np.random` we can also draw samples from probability distributions. For example, we can use `np.random.uniform` (https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.uniform.html#numpy.random.uniform) to draw pseudo-random real numbers from a uniform distribution (https://en.wikipedia.org/wiki/Uniform_distribution_(continuous)).

The code below shows usages of `np.random.uniform`.

```
1  print(np.random.uniform())
2  print(np.random.uniform(low=-1.5, high=2.2))
3  print(repr(np.random.uniform(size=3)))
4  print(repr(np.random.uniform(low=-3.4, high=5.9,
5                               size=(2, 2))))
```

Output   0.756s
```

```
0.7342564226649614
1.2140277197904248
array([0.39686488, 0.35053801, 0.86995248])
array([[-1.53881258,  1.2115093 ],
       [ 5.18860395,  3.17370321]])
```

The function `np.random.uniform` actually has no required arguments. The keyword arguments, `low` and `high`, represent the inclusive lower end and exclusive upper end from which to draw random samples. Since they have default values of 0.0 and 1.0, respectively, the default outputs of `np.random.uniform` come from the range [0.0, 1.0).

The `size` keyword argument is the same as the one for `np.random.randint`, i.e. it represents the output size of the array.

Another popular distribution we can sample from is the normal (Gaussian) distribution (https://en.wikipedia.org/wiki/Normal_distribution). The function we use is `np.random.normal` (https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.normal.html#numpy.random.normal).

The code below shows usages of `np.random.normal`.

```
1  print(np.random.normal())
2  print(np.random.normal(loc=1.5, scale=3.5))
3  print(repr(np.random.normal(loc=-2.4, scale=4.0,
4                              size=(2, 2))))
```

Output   0.453s

```
0.6335194711721529
1.300259818926503
array([[-7.98094753, -2.37502791],
       [-0.47825966, -1.07201821]])
```

Like `np.random.uniform`, `np.random.normal` has no required arguments. The `loc` and `scale` keyword arguments represent the mean and standard deviation, respectively, of the normal distribution we sample from.

NumPy provides quite a few more built-in distributions, which are listed here (https://docs.scipy.org/doc/numpy-1.14.1/reference/routines.random.html).

D. Custom sampling

While NumPy provides built-in distributions to sample from, we can also sample from a custom distribution with the `np.random.choice` (https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.random.choice.html#numpy.random.choice) function.

The code below shows example usages of `np.random.choice`.

```
1  colors = ['red', 'blue', 'green']
2  print(np.random.choice(colors))
3  print(repr(np.random.choice(colors, size=2)))
4  print(repr(np.random.choice(colors, size=(2, 2),
5                              p=[0.8, 0.19, 0.01])))
```

Output   0.508s

```
blue
array(['red', 'green'], dtype='<U5')
array([['blue', 'blue'],
       ['blue', 'red']], dtype='<U5')
```

The required argument for `np.random.choice` is the custom distribution we sample from. The `p` keyword argument denotes the probabilities given to each element in the input distribution. Note that the list of probabilities for `p` must sum to 1.

In the example, we set `p` such that `'red'` has a probability of 0.8 of being chosen, `'blue'` has a probability of 0.19, and `'green'` has a probability of 0.01. When `p` is not set, the probabilities are equal for each element in the distribution (and sum to 1).

Time to Code!

Note: it is important you do all the instructions in the order listed. We test your code by setting a fixed `np.random.seed`, so in order for your code to match the reference output, all the functions must be run in the correct order.

We'll start off by obtaining some random integers. The first integer we get will be randomly chosen from the range [0, 5). The remaining integers will be part of a 3x5 NumPy array, each randomly chosen from the range [3, 10).

Set `random1` equal to `np.random.randint` with `5` as the only argument.

Then set `random_arr` equal to `np.random.randint` with `3` as the first argument, `10` as the `high` keyword argument, and `(3, 5)` as the `size` keyword argument.

```
1   random1 = np.random.randint(5)
2   random_arr = np.random.randint(3, high=10, size=(3, 5))
3
4
```

Show Results   Show Console   X

📋 2 of 2 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓		4	4	Correct value for random1, good job!

Result	Input	Expected Output	Actual Output	Reason
✓		array([[9, 3, 4, 4, 8], [4, 7, 5, …	array([[9, 3, 4, 4, 8], [4, 7, 5, …	Correct value for random_arr, good job!

The next two arrays will be drawn randomly from distributions. The first

0.348s

will contain 5 numbers drawn uniformly from the range [-2.5, 1.5].

Set `random_uniform` equal to `np.random.uniform` with the `low` and `high` keyword arguments set to `-2.5` and `1.5`, respectively. The `size` keyword argument should be set to `5`.

```
1  random_uniform = np.random.uniform(low=-2.5, high=1.5, size=5)
2
3
```

Show Results   Show Console   ✕

📋 1 of 1 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓		array([-2.01740306, -0.55856236, 0.4055 …	array([-2.01740306, -0.55856236, 0.4055 …	Correct value for random_uniform

0.423s

The second array will contain 50 numbers drawn from a normal distribution with mean 2.0 and standard deviation 3.5.

Set `random_norm` equal to `np.random.normal` with the `loc` and `scale` keyword arguments set to `2.0` and `3.5`, respectively. The `size` keyword argument should be set to `(10, 5)`.

```
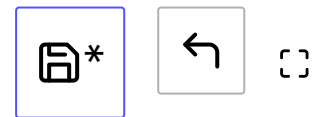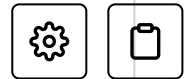1  random_norm = np.random.normal(loc=2.0, scale=3.5, size=(10, 5))
```

```
2
3
```

📋 **1 of 1 Tests Passed**

Result	Input	Expected Output	Actual Output	Reason
✓		array([[1.85833045, -1.6707845 , 3.538 ˮ	array([[1.85833045, -1.6707845 , 3.538 ˮ	Correct value for random_norm

1.162s

We'll now create our own distribution of strings and randomly select from it. The values for our distribution will be `'a'`, `'b'`, `'c'`, `'d'`.

To choose a value, we'll use a probability distribution of `[0.5, 0.1, 0.2, 0.2]`, i.e. `'a'` will have probability 0.5 of being chosen, `'b'` will have a probability of 0.1, etc.

Set `choices` equal to a list of the specified values, in the order given.

Set `choice` equal to `np.random.choice` with `choices` as the first argument and the specified probability distribution as the `p` keyword argument.

```
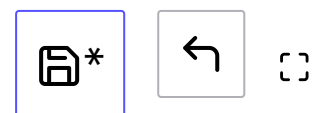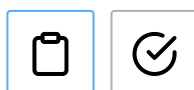1  choices = ['a', 'b', 'c', 'd']
2  choice = np.random.choice(choices, p=[0.5, 0.1, 0.2, 0.2])
3
```

Result	Input	Expected Output	Actual Output	Reason
✓		['a', 'b', 'c', 'd']	['a', 'b', 'c', 'd']	Correct value for choices, good job!
✓		a	a	Correct value for choice, good job!

0.446s

The last random operation we perform will be an in-place shuffle of a NumPy array.

Set `arr` equal to a NumPy array containing the integers from 1 to 5, inclusive.

Then apply `np.random.shuffle` to `arr`.

```
1   arr = np.array([1, 2, 3, 4, 5])
2   np.random.shuffle(arr)
3
```

Show Results   Show Console

1 of 1 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓		array([4, 2, 1, 3, 5])	array([4, 2, 1, 3, 5])	Correct value for arr, good job!

0.348s

✅ **Mark as Completed**

5% completed, meet the <u>criteria</u> and claim your course certificate!

Buy Certificate

⊘ Report an Issue

❓ Ask a Question
(https://discuss.educative.io/tag/random__data-manipulation-with-numpy__machine-learning-for-software-engineers)