4.3 - NumPy Basics: Part 2

Ha Khanh Nguyen (hkfnguyen)

1. Array Indexing & Slicing

1.1 Indexing & slicing with one-dimensional arrays

- One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
import numpy as np

arr = np.arange(10)
arr
```

```
## array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
arr[5]
```

```
## 5
```

```
arr[0:3]
```

```
## array([0, 1, 2])
```

```
arr[0:3] = 12
arr
```

```
## array([12, 12, 12,  3,  4,  5,  6,  7,  8,  9])
```

- An important first distinction from Python's built-in lists is that **array slices are views on the original array**. This means that the data is not copied, and **any modifications to the view will be reflected in the source array**.
- We will test this with the following code segment:

```
arr_slice = arr[5:8]
arr_slice
```

```
## array([5, 6, 7])
```

- Now let’s change the values in `arr_slice` !

```
arr_slice[0] = 12345
arr
```

```
## array([   12,    12,    12,     3,     4, 12345,     6,     7,     8,
##            9])
```

- This is not the case with list at all! (watch the video or try it out yourself)
- **Note**: If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array—for example, `arr[5:8].copy()` .
- The “bare” slice `[:]` will assign to all values in an array:

```
arr_slice[:] = 64
arr
```

```
## array([12, 12, 12,  3,  4, 64, 64, 64,  8,  9])
```

1.2 Indexing & slicing with higer-dimension arrays

- In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d
```

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

- `arr2d[i]` with return the `i+1` -th row of the array in a 2d-array case.

```
arr2d[2]
```

```
## array([7, 8, 9])
```

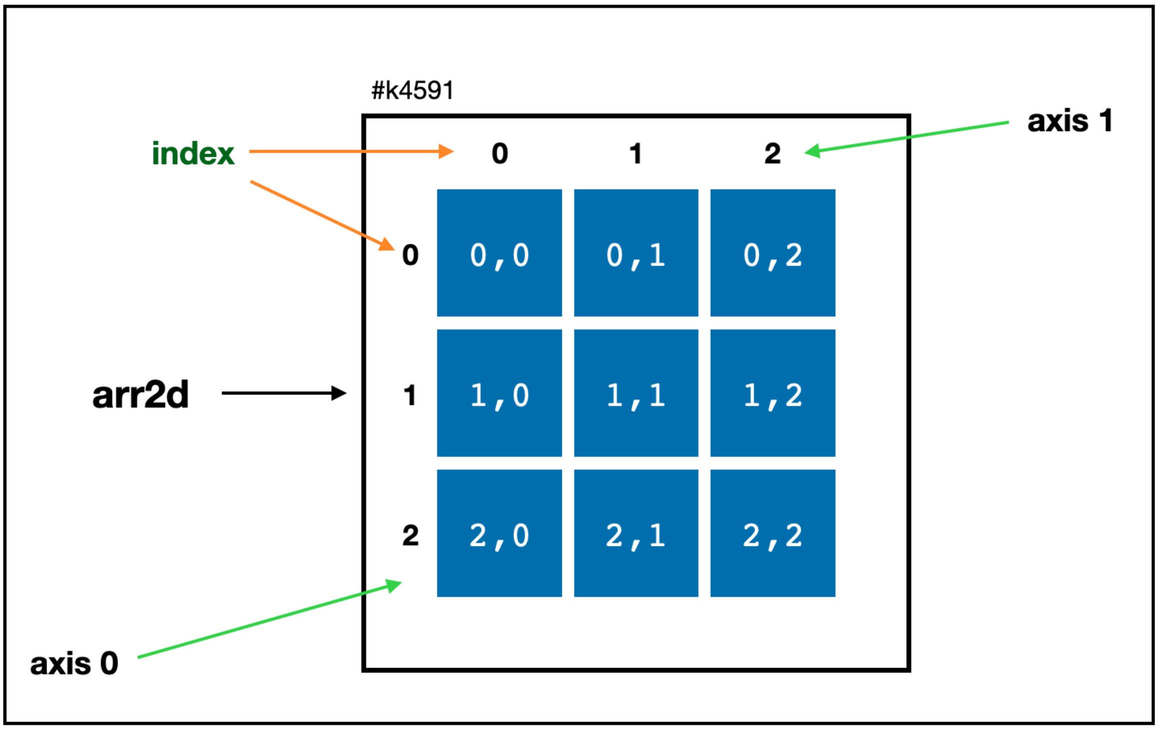
- To select an individual element:

```
arr2d[0, 2]
```

```
## 3
```

```
arr2d[0][2]
```

```
## 3
```



- Now, let’s try slicing with 2d-array!

```
arr2d
```

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

```
arr2d[:2] # same as arr2d[0:2]
```

```
## array([[1, 2, 3],
##        [4, 5, 6]])
```

- What if we want to slice along axis 1 only!

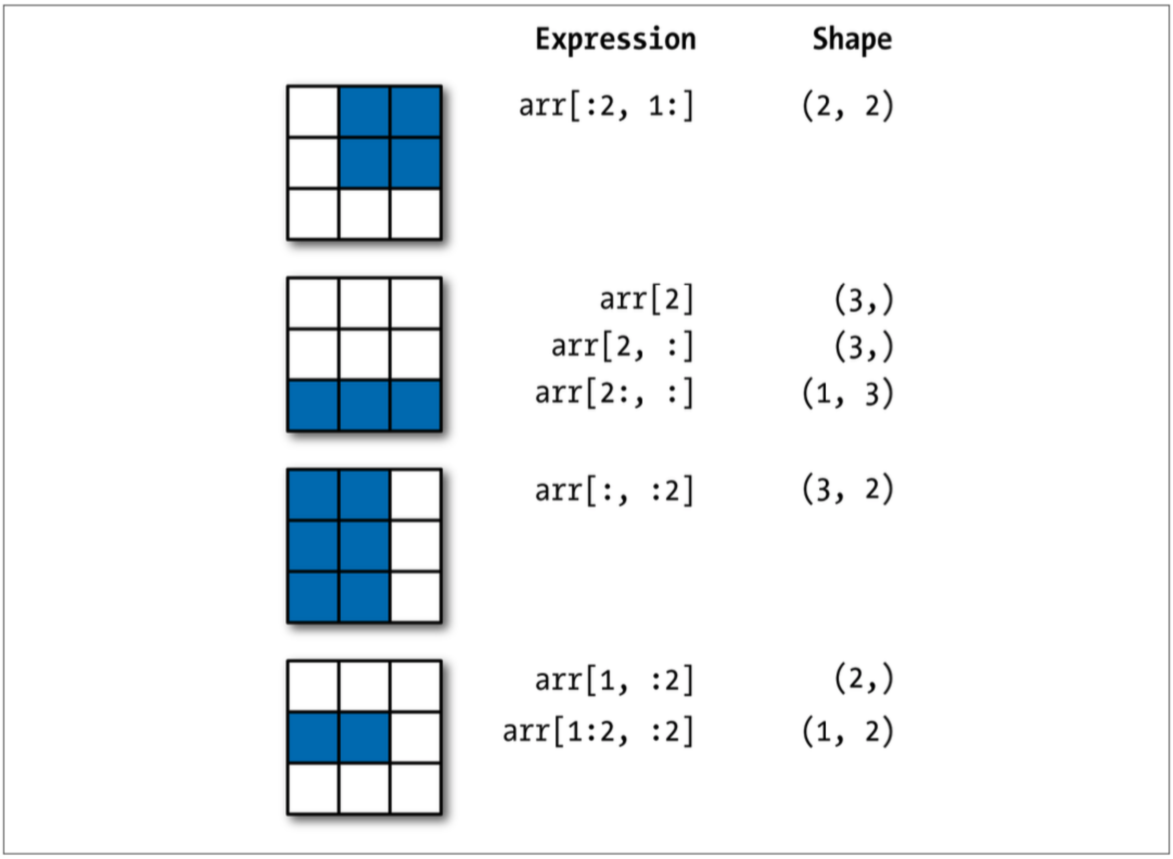
```
arr2d[:, 1:]
```

```
## array([[2, 3],
##        [5, 6],
##        [8, 9]])
```

- When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices.

```
arr2d[1, :2]
```

```
## array([4, 5])
```



- Note that assigning values to a slice expression assigns to the whole selection:

```
arr2d[:2, 1:] = 0
arr2d
```

```
## array([[1, 0, 0],
##        [4, 0, 0],
##        [7, 8, 9]])
```

1.3 Boolean Indexing

- Let's consider an example where we have some data in an array and an array of names with duplicates.

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
names
```

```
## array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

- Now, we will generate some random normally distributed data (more on this later):

```
# set seed
np.random.seed(430)

data = np.random.randn(7, 4)
data
```

```
## array([[ 0.4742312 , -0.85940424, -0.75509958, -0.54634703],
##        [ 1.01367802, -1.48055291,  2.12899127, -0.20291703],
##        [ 0.57428592, -0.35479235,  0.19028642,  1.10250873],
##        [ 1.24961928,  0.92632742, -0.45853894,  0.65669902],
##        [-0.25024139,  0.16887152,  0.05787939, -0.40472773],
##        [-0.99232288,  1.17977178, -0.15407498, -1.02124231],
##        [ 0.95328187,  1.64033079,  0.68565206,  0.54643213]])
```

- Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name `'Bob'` .

```
names == 'Bob'
```

```
## array([ True, False, False,  True, False, False, False])
```

```
data[names == 'Bob']
```

```
## array([[ 0.4742312 , -0.85940424, -0.75509958, -0.54634703],
##        [ 1.24961928,  0.92632742, -0.45853894,  0.65669902]])
```

- Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like `&` (and) and `|` (or):

```
data[(names == 'Bob') | (names == 'Will')]
```

```
## array([[ 0.4742312 , -0.85940424, -0.75509958, -0.54634703],
##        [ 0.57428592, -0.35479235,  0.19028642,  1.10250873],
##        [ 1.24961928,  0.92632742, -0.45853894,  0.65669902],
##        [-0.25024139,  0.16887152,  0.05787939, -0.40472773]])
```

- Setting values with boolean arrays works in a common-sense way. To set all of the negative values in data to 0 we need only do:

```
data[data < 0] = 0
```

2. Universal Functions: Fast Element-wise Array Functions

- A universal function, or *ufunc*, is a function that performs element-wise operations on data in ndarrays.
- Many ufuncs are simple element-wise transformations, like `sqrt` or `exp` :

```
arr = np.arange(10)
arr
```

```
## array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
# square root
np.sqrt(arr)
# exponential
```

```
## array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
##        2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])
```

```
np.exp(arr)
```

```
## array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
##        5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
##        2.98095799e+03, 8.10308393e+03])
```

- These are referred to as **unary ufuncs**.
- Others, such as `add` or `maximum`, take two arrays (thus, **binary ufuncs**) and return a single array as the result:

```
x = np.random.randn(5)
y = np.random.randn(5)
x
```

```
## array([-2.76931098, -0.30328563, -0.38740235,  0.55320402, -1.03859445])
```

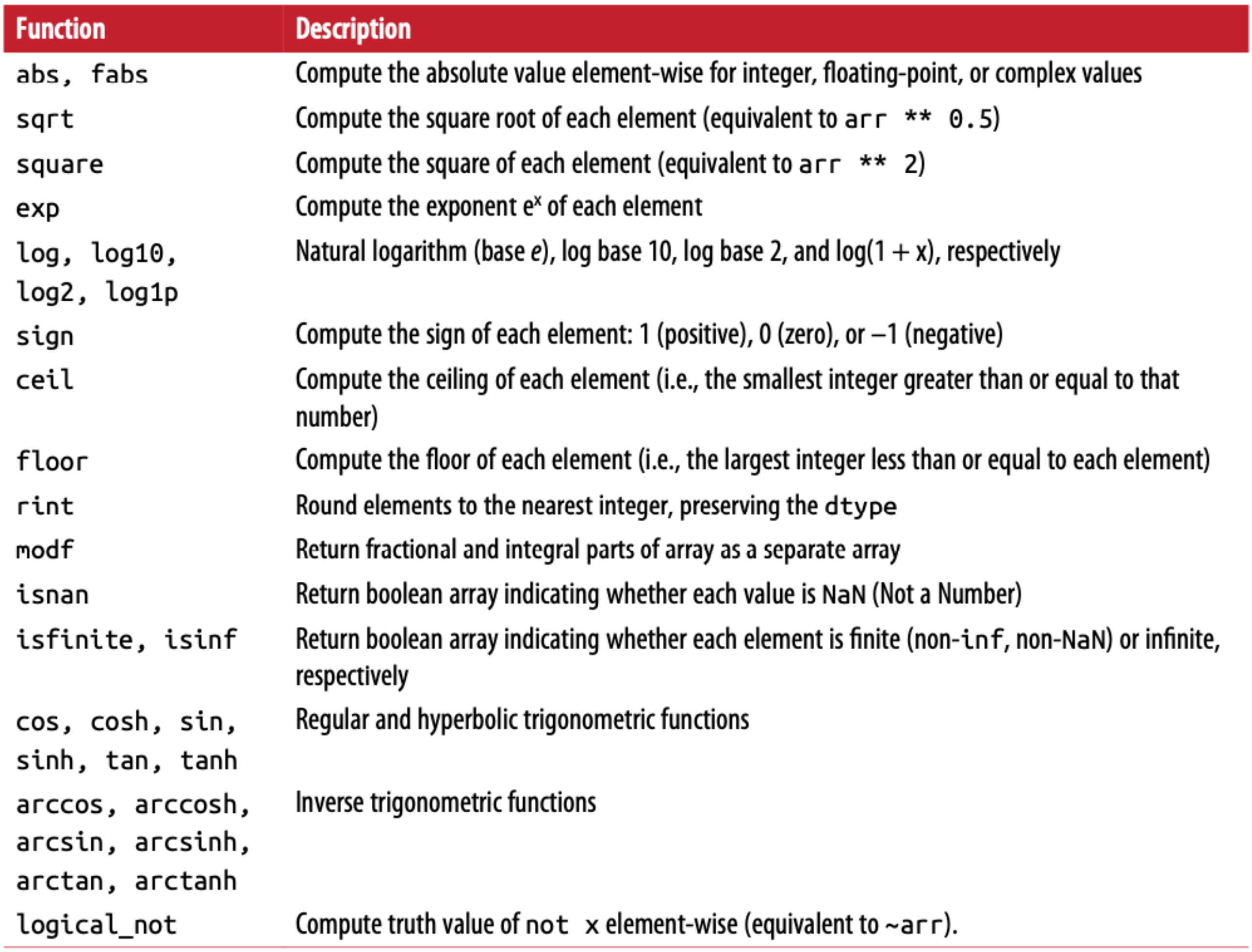
```
y
```

```
## array([-0.00931142, -1.24925273, -0.90224748, -0.97430454, -1.04522033])
```

```
np.maximum(x, y)
```

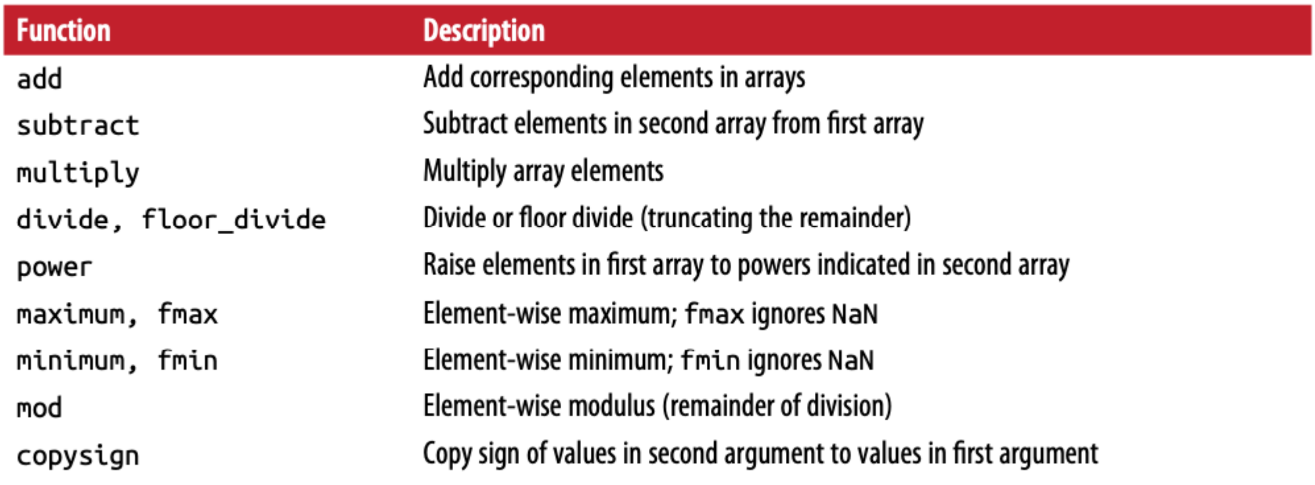
```
## array([-0.00931142, -0.30328563, -0.38740235,  0.55320402, -1.03859445])
```

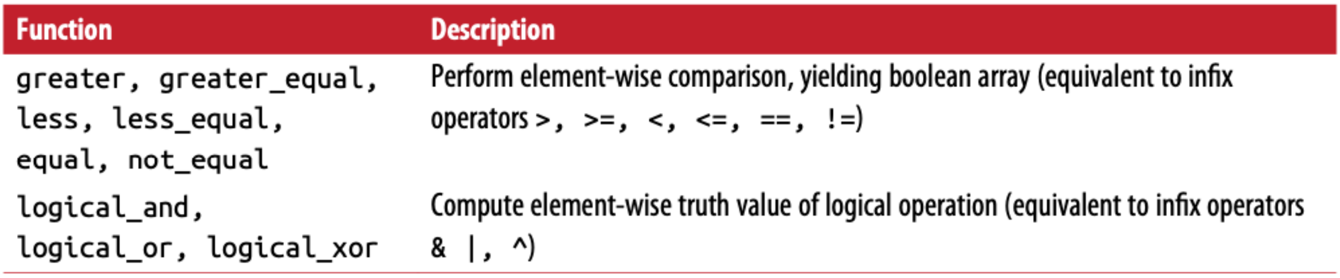
Table 4-3. Unary ufuncs



Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
`ceil`	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as a separate array
`isnan`	Return boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise (equivalent to `~arr`).

Table 4-4. Binary universal functions





Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum; `fmax` ignores `NaN`
`minimum, fmin`	Element-wise minimum; `fmin` ignores `NaN`
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array (equivalent to infix operators `>, >=, <, <=, ==, !=`)
`logical_and, logical_or, logical_xor`	Compute element-wise truth value of logical operation (equivalent to infix operators `& |, ^`)

Exercise

Convert the following code segment from built-in Python list to NumPy arrays:

```
nums = np.random.randn(10)
new_nums = []
for x in nums:
  if x > 0:
    new_nums.append(np.sqrt(x))
  else:
    new_nums.append(-x)
```

3. Pseudorandom Number Generator

- The `numpy.random` module supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions:

```
samples = np.random.normal(size=(4,4))
samples
```

```
## array([[ 0.25802529,  0.33110554, -0.80587636,  0.01472604],
##        [ 0.87316655, -0.53277903,  2.3312076 , -0.32409937],
##        [ 0.04670989, -0.97748728,  0.24504297, -0.49180966],
##        [ 0.04692382,  0.88000198, -0.79238475, -1.50101697]])
```

- We say that these are **pseudorandom** numbers because they are generated by an algorithm with deterministic behavior based on the seed of the random number generator.
- You can change NumPy's random number generation seed using `np.random.seed` :

```
np.random.seed(430)
```

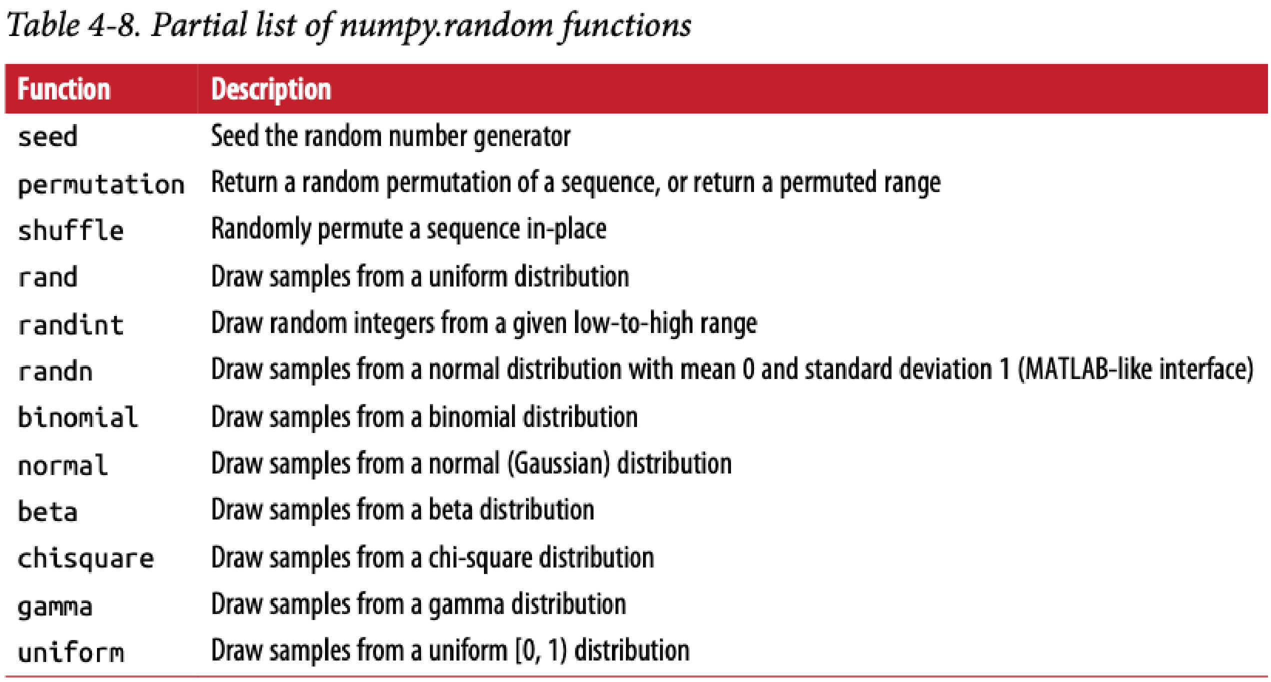


Table 4-8. Partial list of numpy.random functions

Function	Description
`seed`	Seed the random number generator
`permutation`	Return a random permutation of a sequence, or return a permuted range
`shuffle`	Randomly permute a sequence in-place
`rand`	Draw samples from a uniform distribution
`randint`	Draw random integers from a given low-to-high range
`randn`	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
`binomial`	Draw samples from a binomial distribution
`normal`	Draw samples from a normal (Gaussian) distribution
`beta`	Draw samples from a beta distribution
`chisquare`	Draw samples from a chi-square distribution
`gamma`	Draw samples from a gamma distribution
`uniform`	Draw samples from a uniform [0, 1) distribution

This lecture note is modified from Chapter 4 of Wes McKinney's Python for Data Analysis 2nd Ed (https://www.oreilly.com/library/view/python-for-data/9781491957653/).