Python and Transforms

Some Tensor Manipulation

Alexander G. Ororbia II

COGS-621: Foundations of Scientific Computing

9/9/2025

Creating Different Kinds of (Initial) Arrays

- Useful ways to create (instantiate) ndarrays pre-filled with particular values

np.eye :
Creates diagonal array with values on main diagonal

Function Name	Type of Array
`np.array`	Create an array for which the elements are given by an array-like object, which, for example, can be a (nested) Python list, a tuple, an iterable sequence, or another `ndarray` instance.
`np.zeros`	Create an array with the specified dimensions and data type that is filled with zeros.
`np.ones`	Create an array with the specified dimensions and data type that is filled with ones.
`np.diag`	Create a diagonal array with specified values along the diagonal and zeros elsewhere.
`np.arange`	Create an array with evenly spaced values between the specified start, end, and increment values.
`np.linspace`	Create an array with evenly spaced values between specified start and end values, using a specified number of elements.
`np.logspace`	Create an array with values that are logarithmically spaced between the given start and end values.
`np.meshgrid`	Generate coordinate matrices (and higher-dimensional coordinate arrays) from one-dimensional coordinate vectors.
`np.fromfunction`	Create an array and fills it with values specified by a given function, which is evaluated for each combination of indices for the given array size.
`np.fromfile`	Create an array with the data from a binary (or text) file. NumPy also provides a corresponding function `np.tofile` with which NumPy arrays can be stored to disk and later read back using `np.fromfile`.
`np.genfromtxt`, `np.loadtxt`	Create an array from data read from a text file, for example, a comma-separated value (CSV) file. The `np.genfromtxt` function also supports data files with missing values.
`np.random.rand`	Generate an array with random numbers that are uniformly distributed between 0 and 1. Other types of distributions are also available in the `np.random` module.

```
In [10]: np.array([1, 2, 3], dtype=int)
Out[10]: array([1, 2, 3])
In [11]: np.array([1, 2, 3], dtype=float)
Out[11]: array([ 1.,  2.,  3.])
In [12]: np.array([1, 2, 3], dtype=complex)
Out[12]: array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

```
In [13]: data = np.array([1, 2, 3], dtype=float)
In [14]: data
Out[14]: array([ 1.,  2.,  3.])
In [15]: data.dtype
Out[15]: dtype('float64')
In [16]: data = np.array(data, dtype=int)
In [17]: data.dtype
Out[17]: dtype('int64')
In [18]: data
Out[18]: array([1, 2, 3])
```

```
In [19]: data = np.array([1, 2, 3], dtype=float)
In [20]: data
Out[20]: array([ 1.,  2.,  3.])
In [21]: data.astype(int)
Out[21]: array([1, 2, 3])
```

```
In [51]: np.arange(0.0, 11, 1)
Out[51]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9. ,  10.])
In [52]: np.linspace(0, 10, 11)
Out[52]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  10.])
```

Distributions and sampling: np.random

- Distributions over random variables (numbers) will be useful later on
  - Some useful (continuous) ones: uniform ($U(a, b)$), normal (Gaussian; $N(\mu, \sigma)$), laplace (Laplacian; $L(loc, scale)$)
  - Usually take in distribution-specific parameters and then a "size" argument (which is the shape of the tensor you want); if nothing but size is provided, then you often get samples of the standard distribution
- These routines are useful for sampling basic distributions or composing others in terms of basic ones

```
>>> np.random.uniform
```

```
>>> np.random.normal
```

```
>>> np.random.laplace
```

Slicing and accessing

Table 2-4. *Examples of Array Indexing and Slicing Expressions*

Expression	Description
`a[m]`	Select the element at index *m*, where *m* is an integer (start counting form 0).
`a[-m]`	Select the *n*th element from the end of the list, where *m* is an integer. The last element in the list is addressed as –1, the second to last element as –2, and so on.
`a[m:n]`	Select elements with index starting at *m* and ending at *n* – 1 (*m* and *n* are integers).
`a[:]`	Select all elements in the given axis.
`a[:n]`	Select elements starting with index 0 and going up to index *n* – 1 (integer).
`a[m:]`	Select elements starting with index *m* (integer) and going up to the last element in the array.
`a[m:n:p]`	Select elements with index *m* through *n* (exclusive), with increment *p*.
`a[::-1]`	Select all the elements, in reverse order.

Knowing what “views” are

Using a view to your advantage (using a sub-view to set values):

```
In [85]: B = A[1:5, 1:5]
In [86]: B
Out[86]: array([[11, 12, 13, 14],
                [21, 22, 23, 24],
                [31, 32, 33, 34],
                [41, 42, 43, 44]])
In [87]: B[:, :] = 0
In [88]: A
Out[88]: array([[ 0,  1,  2,  3,  4,  5],
                [10,  0,  0,  0,  0, 15],
                [20,  0,  0,  0,  0, 25],
                [30,  0,  0,  0,  0, 35],
                [40,  0,  0,  0,  0, 45],
                [50, 51, 52, 53, 54, 55]])
```

When you do not want to use a view (extract sub-array as a “clone”):

```
In [89]: C = B[1:3, 1:3].copy()
In [90]: C
Out[90]: array([[0, 0],
                [0, 0]])
In [91]: C[:, :] = 1  # this does not affect B since C is a copy of the view B[1:3, 1:3]
In [92]: C
Out[92]: array([[1, 1],
                [1, 1]])
In [93]: B
Out[93]: array([[0, 0, 0, 0],
                [0, 0, 0, 0],
                [0, 0, 0, 0],
                [0, 0, 0, 0]])
```

Reshaping and resizing arrays

Table 2-5. Summary of NumPy Functions for Manipulating the Dimensions and the Shape of Arrays

Function/Method	Description
`np.reshape`, `np.ndarray.reshape`	Reshape an N-dimensional array. The total number of elements must remain the same.
`np.ndarray.flatten`	Create a copy of an N-dimensional array and reinterprets it as a one-dimensional array (i.e., all dimensions are collapsed into one).
`np.ravel`, `np.ndarray.ravel`	Create a view (if possible, otherwise a copy) of an N-dimensional array in which it is interpreted as a one-dimensional array.
`np.squeeze`	Remove axes with length 1.
`np.expand_dims`, `np.newaxis`	Add a new axis (dimension) of length 1 to an array, where `np.newaxis` is used with array indexing.
`np.transpose`, `np.ndarray.transpose`, `np.ndarray.T`	Transpose the array. The transpose operation corresponds to reversing (or, more generally, permuting) the axes of the array.
`np.hstack`	Stack a list of arrays horizontally (along axis 1): for example, given a list of column vectors, it appends the columns to form a matrix.
`np.vstack`	Stack a list of arrays vertically (along axis 0): for example, given a list of row vectors, it appends the rows to form a matrix.
`np.dstack`	Stack arrays depth-wise (along axis 2).
`np.concatenate`	Create a new array by appending arrays after each other along a given axis.
`np.resize`	Resize an array. Create a new copy of the original array, with the requested size. If necessary, the original array is repeated to fill up the new array.
`np.append`	Append an element to an array. Create a new copy of the array.
`np.insert`	Insert a new element at a given position. Create a new copy of the array.
`np.delete`	Delete an element at a given position. Create a new copy of the array.

Know that reshaping yields views

```
In [112]: data = np.array([[1, 2], [3, 4]])
In [113]: np.reshape(data, (1, 4))
Out[113]: array([[1, 2, 3, 4]])
In [114]: data.reshape(4)
Out[114]: array([1, 2, 3, 4])
```

```
In [115]: data = np.array([[1, 2], [3, 4]])
In [116]: data
Out[116]: array([[1, 2],
                 [3, 4]])
In [117]: data.flatten()
Out[117]: array([ 1,  2,  3,  4])
In [118]: data.flatten().shape
Out[118]: (4,)
```

Questions?