

## ✓ Chapter 8 - Multidimensional NumPy Arrays

### Goals

- Create, use and modify multidimensional NumPy arrays (2D, 3D, etc.)
- Apply functions to multidimensional arrays
- Read in a comma-separated-value (csv) data file

## 8.0 Multidimensional Arrays

This chapter describes how to create, apply and extract data from multi-dimensional arrays in 2D, 3D and higher dimensions.

## ✓ 8.1 Creating multidimensional arrays

### 8.1.1 Creating 2D arrays

- 2D Arrays are specified by an `(n,m)` tuple
- The first parameter `n` specifies the number rows
- The second parameter `m` specifies the number of columns.

## ✓ 🌟 Example: Creating 2D array filled with zeros


- To fill a  $2 \times 3$  array (2 rows and 3 columns) with zeros, we can use the `np.zeros((n,m))` command.
- The "double parentheses" are necessary, since the inner parentheses create a "tuple" which the `np.zeros()` function expects.

```
import numpy as np

A = np.zeros((2,3))
print(A)
```

## ✓ 🌟 Example: Creating an array of random integers

In the following example, we fill a 2x5 array with random integers between 1 and 9 inclusive.

- We specify the size of the array to have 3 rows and 5 columns using the `size=(3,5)` parameter.
- We use the `endpoint=True` parameter  to tell Python to include the upper limit (9 in this example).

```
rng = np.random.default_rng()      # create random number Generator

A = rng.integers(1, 9, size=(3,5),endpoint=True)
print(A)
```

## ✓ Skill Check 1

Create a 2D array with 6 columns and 3 rows. Each element should be a random integer whose value satisfies  $1 \leq x \leq 4$ . Print the array.

```
# Your code here
```

## ✓ 8.1.2 Creating 3D arrays

- 3D arrays (i.e. tensors) are specified by a `(p,n,m)` tuple
- `p` = first parameter = the number of layers p of the array.
- `n` = 2nd parameter = the number rows n
- `m` = 3rd parameter = the number of columns m
- For example, to fill an 4x2x3 array (2 rows, 3 columns and 4 layers deep) with zeros, we can use the following:

```
A = np.zeros((4,2,3))
print(A)
```

## ✓ 8.2 .shape, .size and .ndim attributes

NumPy provides three attributes that characterize the shape and size of NumPy arrays. Remember that attributes do not use parentheses, while methods do.

- `.shape` returns a tuple containing the dimensions along each axis of the array.
- The number of elements along a given axis can be extracted as follows:
- The first element is the number of rows: `.shape[0]`
- The second element is the number of columns: `.shape[1]`
- Higher-dimensional arrays (3D, 4D, etc.) will have information about the number of layers in the `[2]`, `[3]`, etc. positions.

```
A = rng.integers(1, 9, size=(3,5),endpoint=True)
print("A = ")
print(A)
```

```
print()
print("shape of A is ",A.shape)

n_rows = A.shape[0]          # extract the number of rows
n_cols = A.shape[1]          # extract the number of columns

print("number of rows in A   =",n_rows)
print("number of columns in A =",n_cols)
```

- `.size` returns the total number of elements in A, which equals the (number of rows) × (number of columns) for a 2D array
- `.ndim` returns the number of dimensions, which is 2 for a 2D array, 3 for a 3D array, etc.

```
print("size of A (total # elements) =",A.size)
print("number of dimensions of A   =",A.ndim)
```

## ☀ Example: 4D Array

Here's an example of creating a 4D array:

- number of columns = 5
- number of rows = 4
- number of layers = 3
- number of blocks of layers = 2

```
z = np.zeros((2,3,4,5))

print("shape of z (length along each axis) =",z.shape)
print("size of z (total # elements) =",z.size)
print("number of dimensions of A   =",z.ndim)
print()
print("z = ")
print(z)
```

## 8.4 Accessing elements in multi-dimensional arrays

The following examples show how to access and slice a 2D array.

```
rng = np.random.default_rng()  # create random number Generator
A = rng.integers(1, 9, size=(3,5),endpoint=True)

print("A = ")
print(A)          # prints array A
print()           # prints a blank line

print("A[0,2] = ",A[0,2])    # top row, 3rd column
print("A[0,:2] = ",A[0,1:3]) # top row, columns 2 and 3
print("A[0,:] = ",A[0,:])    # top row
print("A[0,:] = ",A[:,1])    # second column
```

```
print()
print("A[0:2,0:2] = ")          # 2x2 matrix from top-left corner of A
print(A[0:2,0:2])
```

The following over-writes the top row of our 2D array with zeros.

- Note: we use the `.shape` attribute to automatically determine how many columns are in `A`, which we pass to the `np.zeros()` function.

```
A[0,:] = np.zeros(A.shape[1])
print(A)
```

## ✓ Skill Check 2

- Use slicing to set the last column of the `A` array defined above to 1's
- print the `A` array

```
# your code here
```

## ✓ 8.5 Working with multi-dimensional arrays

### ✓ 8.5.1 Create a new array to match size of an existing array

- We can use the `.shape` attribute to easily create a new array with the same size as an existing array like this:

```
A = rng.integers(1, 9, size=(3,5),endpoint=True)

B = np.zeros(A.shape)    # Array B will have same size as array A
print(B)
```

### ✓ 8.5.2 Take the transpose of a 2D array

- The transpose flips the rows and columns
- We just add a `.T` after our array like this: `A.T`
- We'll take the transpose of the `A` array in the previous section:

```
print("original Array, A:")
print(A)
print()

print("transpose of A:")
print(A.T)
```

### 8.5.3 Reshaping arrays

#### Flatten

- We can reduce an array to 1D with the `.flatten()` method.
- Notice that

```
print("flattened version of A")
A_flat = A.flatten()
print(A_flat)
```

#### Reshape

- The `.reshape()` method can be used to reorder the elements of an array.
- The `.reshape()` method takes a tuple as an argument: single number for 1D array, `(n,m)` for 2D array, `(p,n,m)` for a 3D array and so on.

```
##### Create a 1x12 array filled with integers
print("original array. A ")
A = np.linspace(1,12,12,dtype=int)
print(A)

##### Reshape -> 2x6 (2 rows, 6 columns)
print()
print("A2 = A.reshape((2,6)) ")
A2 = A.reshape((2,6))
print(A2)
print()

##### Reshape -> 3x4 (3 rows, 4 columns)
print("A3 = A2.reshape((3,4)) ")
A3 = A2.reshape((3,4))
print(A3)
print()

##### Reshape -> 2x2x3 (2 layers, 2 rows, 3 columns) = 3D array
print("A4 = A3.reshape((2,2,3)) ")
A4 = A3.reshape((2,2,3))
print(A4)
```

### 8.5.4 Analyzing Arrays along Rows or Columns

- Suppose you are given a 2D array, and you'd like to know something about the first column.

### 8.5.4 Computing statistics along rows or columns

- Suppose you are given a 2D array, and you'd like to know something about the first column.

- We can use slicing to extract the column for analysis
- We can also specify the dimension we want to analyze

```
R = np.linspace(1,15,15)
R = R.reshape((5,3))
R = R.T
print("R = ")
print(R)

ave = R.mean(axis = 0)
print()
print("Column mean = ")
print(ave)

ave = R.mean(axis = 1)
print()
print("Row mean = ")
print(ave)
```

## ✓ 8.6 Reading data files in Google Colab

Google Colab poses some challenges when reading and writing files due to the way Google handles directories. When we run Python on your laptop, this process will be much simpler.

- Step 1 - Download the file `bh_merger.csv` from the `Python_Tutorials` directory on GitHub and save it on your laptop. This file contains the masses of binary black hole pairs that have merged and given off gravitational waves.
- Step 2 - Run the following Code Cell. This will load in the data in from the file and save it as a 2D NumPy array called `data`.
- Step 3 - Once the data is successfully read in to the `data` array, the data will be saved in memory, so you can run or rerun the analysis code in the following Code Cells without having to reload the file.

Here's what the following Code Cell does to read in the data file:

- Import the `google.files` module
- Call `uploaded = files.upload()` to launch a file picker. You'll need to navigate to you the `bh_merger.csv` file to load it.
- Extract the actual file name from the `uploaded` dictionary using the command `file_name = list(uploaded.keys())[0]`
- Finally, we can now use the NumPy command `np.loadtxt(file_name)` to load a file specified by `file_name`.

Use of the `np.loadtxt()` command

- `np.loadtxt(file_name)` loads a data file with name `file_name`. The file must contain rows of numeric values with equal number of values per row, with each numerical value separated

by spaces. The file cannot have any extra header or footer data. Thus, the file must have the form:

```
1.0  2.4  3.6
3.5  0.1  0.4
4.3  2.1  3.5
```

- If the values are separated by commas as in the following example, we need to specify the delimiter between values is a comma like this: `np.loadtxt(file_name,delimiter=',')`.

```
1.0, 2.4, 3.6
3.5, 0.1, 0.4
4.3, 2.1, 3.5
```

- If the file has one or more header lines, we need to skip over them to read the data. We specify the number of lines to skip with the `skiprows` parameter like this: `np.loadtxt(file_name, skiprows=1,delimiter=',')`.

```
x, y, z
1.0, 2.4, 3.6
3.5, 0.1, 0.4
4.3, 2.1, 3.5
```

- Our `bh_merger.csv` file has one header line and uses a comma delimiter, so we use the command `data = np.loadtxt(file_name,skiprows=1,delimiter=',')` to read it.

```
# Import Libraries
from google.colab import files # used to load files in Google
import numpy as np

##### Load Data File
uploaded = files.upload() # This opens a file picker

# uploaded is a dictionary: filename -> bytes
# Get the first filename
file_name = list(uploaded.keys())[0]

# Load the data into a NumPy array
# Assuming a text file with numeric data, separated by commas
# and with one header line that must be skipped
data = np.loadtxt(file_name,skiprows=1,delimiter=',')
```

The `bh_merger.csv` file contains the following columns of data:

- column 1 - mass of first black hole in units of solar masses (i.e. 30 means the black hole has 30 times the mass of the sun).
- column 2 - mass of the second black hole

- column 3 - distance of the black holes from Earth in units of megaparsecs (Mpc). 1 Mpc = 3.26 million light years.

The following Code prints the number of rows, columns and first 5 rows of data.

```
# Inspect the array
nrows = data.shape[0]
ncols = data.shape[1]
print()
print(f"{file_name} contains {nrows} rows and {ncols} columns")
print()
print("First 5 rows:")
print(data[:5])
```

### ✓ Skill Check 3

The following code shows how to extract the first column of values from the `data` array and name it `mass1` (this will contain the masses of one of black holes in the binary system.)

- Extract the second column and name it `mass2` (mass of second black hole)
- Extract the third column and name it `dist` (distance of the black holes)

```
mass1 = data[:,0] # mass one is stored in the first column (index = 0)

# your code here
```

### ✓ Skill Check 4

- Calculate and print the minimum, maximum and average mass for `mass1`. Use the NumPy vectorized methods
- Calculate and print the minimum, maximum and average mass for `mass2`. Use the NumPy vectorized methods
- Calculate and print the minimum, maximum and average distance of the black holes. Use the NumPy vectorized methods.
- Convert the distance of the most distant black hole into billions of light years (multiply the distance in Mpc by 3262). For nearby objects, this would be how far back in the past light left the object that we are now seeing. However, these distant black holes are so far away, we would need to take into account the acceleration of the universe after the Big Bang, which only happened about 13.8 billion years ago.

```
# your code here
```



## ✓ Skill Check 5

- Reshape the the first two columns of the the `data` array to create a single array containing the masses of both black holes.
- Calculate the minimum, maximum and average of all the black holes.
- Are your values consistent with your stats for `mass1` and `mass2`?

```
# your code here
```

## ✓ Key Points

- NumPy arrays can be 1D, 2D, 3D, 4D, or any number of dimensions
- Fetching values from a multidimensional array uses commas, i.e `A[1,0,3]`
- NumPy arrays have many Attributes an Methods to perform operations and gather statistics.
- Loading data into Google Colab is more complicated than doing so on your laptop. We showed how to load a comma-separated-value (csv) file and analyze the data

This tutorial is an adaptation of "[Python for Physicists](#)" © [Software Carpentry](#)