Getting Started with NumPy

Introduction

NumPy is one of the main libraries for performing scientific computing in Python. Using NumPy, you can create high-performance multi-dimensional arrays, and several tools to work with these arrays.

A NumPy array can store a grid of values. All the values must be of the same type. NumPy arrays are n-dimensional, and the number of dimensions is denoted by the *rank* of the NumPy array. The shape of an array is a tuple of integers which holds the size of the array along each of the dimensions.

For more information on NumPy, refer to http://www.numpy.org/ (http://www.numpy.org/).

Objectives

You will be able to:

- Use broadcasting to perform a math operation on an entire numpy array
- Perform vector and matrix operations with numpy
- Access the shape of a numpy array
- Use indexing with numpy arrays

NumPy array creation and basic operations

First, remember that it is customary to import NumPy as `np`.

In [1]:
```
import numpy as np
```

One easy way to create a numpy array is from a Python list. The two are similar in a number of manners but NumPy is optimized in a number of ways for performing mathematical operations, including having a number of built-in methods that will be extraordinarily useful.

In [8]:
```
x = np.array([1, 2, 3])
print(type(x))
```

```
<class 'numpy.ndarray'>
```

Broadcasting Mathematical Operations

Notice right off the bat how basic mathematical operations will be applied elementwise in a NumPy array versus a literal interpretation with a Python list:

In [9]:
```
# Multiplies each element by 3
x * 3
```

Out[9]:
```
array([3, 6, 9])
```

In [10]:
```
# Returns the list 3 times
[1, 2, 3] * 3
```

Out[10]:
```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

In [11]:
```
# Adds two to each element
x + 2
```

Out[11]:
```
array([3, 4, 5])
```

In [12]:
```
# Returns an error; different data types
[1, 2, 3] + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_213/3278319400.py in <module>
      1 # Returns an error; different data types
----> 2 [1, 2, 3] + 2

TypeError: can only concatenate list (not "int") to list
```

Even more math!

Scalar Math

`np.add(arr,1)`	Add 1 to each array element
`np.subtract(arr,2)`	Subtract 2 from each array element
`np.multiply(arr,3)`	Multiply each array element by 3
`np.divide(arr,4)`	Divide each array element by 4 (returns `np.nan` for division by zero)
`np.power(arr,5)`	Raise each array element to the 5th power

Vector Math

`np.add(arr1,arr2)`	Elementwise add arr2 to arr1
`np.subtract(arr1,arr2)`	Elementwise subtract arr2 from arr1
`np.multiply(arr1,arr2)`	Elementwise multiply arr1 by arr2
`np.divide(arr1,arr2)`	Elementwise divide arr1 by arr2
`np.power(arr1,arr2)`	Elementwise raise arr1 raised to the power of arr2
`np.array_equal(arr1,arr2)`	Returns True if the arrays have the same elements and shape

`np.sqrt(arr)`	Square root of each element in the array
`np.sin(arr)`	Sine of each element in the array
`np.log(arr)`	Natural log of each element in the array
`np.abs(arr)`	Absolute value of each element in the array
`np.ceil(arr)`	Rounds up to the nearest int
`np.floor(arr)`	Rounds down to the nearest int
`np.round(arr)`	Rounds to the nearest int

Here's a few more examples from the list above

In [7]:
```
# Adding raw lists is just appending
[1, 2, 3] + [4, 5, 6]
```

Out[7]:
```
[1, 2, 3, 4, 5, 6]
```

In [8]:
```
# Adds elements
np.array([1, 2, 3]) + np.array([4, 5, 6])
```

Out[8]:
```
array([5, 7, 9])
```

In [9]:
```
# Same as above with built-in method
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
np.add(x, y)
```

Out[9]:
```
array([5, 7, 9])
```

Multidimensional Arrays

NumPy arrays are also very useful for storing multidimensional data such as matrices. Notice how NumPy tries to nicely align the elements.

In [13]:
```
# An ordinary nested list
y = [[1, 2], [3, 4]]
print(type(y))
y
```

```
<class 'list'>
```

Out[13]:
```
[[1, 2], [3, 4]]
```

In [14]:
```
# Reformatted as a NumPy array
y = np.array([[1, 2], [3, 4]])
print(type(y))
y
```

```
<class 'numpy.ndarray'>
```

Out[14]:
```
array([[1, 2],
       [3, 4]])
```

The Shape Attribute

One of the most important attributes to understand with this is the shape of a NumPy array.

In [15]:
```
y.shape
```

Out[15]:
```
(2, 2)
```

In [13]:
```
y = np.array([[1, 2, 3],[4, 5, 6]])
print(y.shape)
y
```

```
(2, 3)
```

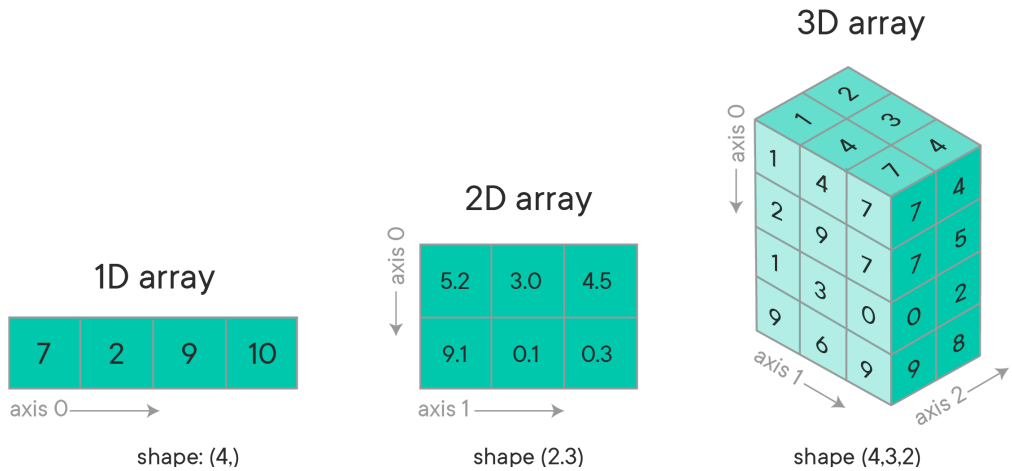
Out[13]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [16]:
```
y = np.array([[1, 2],[3, 4],[5, 6]])
print(y.shape)
y
```

```
(3, 2)
```

Out[16]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

We can also have higher dimensional data such as working with 3 dimensional data



In [17]:
```
y = np.array([[[1, 2],[3, 4],[5, 6]],
              [[1, 2],[3, 4],[5, 6]]
              ])
print(y.shape)
y
```

```
(2, 3, 2)
```

Out[17]:
```
array([[[1, 2],
        [3, 4],
        [5, 6]],

       [[1, 2],
        [3, 4],
        [5, 6]]])
```

Built-in Methods for Creating Arrays

NumPy also has several built-in methods for creating arrays that are useful in practice. These methods are particularly useful:

- `np.zeros(shape)`
- `np.ones(shape)`
- `np.full(shape, fill)`

In [16]:
```
# One dimensional; 5 elements
np.zeros(5)
```

Out[16]:
```
array([0., 0., 0., 0., 0.])
```

In [17]:
```
# Two dimensional; 2x2 matrix
np.zeros([2, 2])
```

Out[17]:
```
array([[0., 0.],
       [0., 0.]])
```

In [18]:
```
# 2 dimensional;  3x5 matrix
np.zeros([3, 5])
```

Out[18]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

In [19]:
```
# 3 dimensional; 3 4x5 matrices
np.zeros([3, 4, 5])
```

Out[19]:
```
array([[[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]],

       [[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]],

       [[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]])
```

Similarly the `np.ones()` method returns an array of ones

In [20]:
```
np.ones(5)
```

Out[20]:
```
array([1., 1., 1., 1., 1.])
```

In [21]:
```
np.ones([3, 4])
```

Out[21]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

The `np.full()` method allows you to create an array of arbitrary values

In [22]:
```
# Create a 1d array with 5 elements, all of which are 3
np.full(5, 3)
```

Out[22]:
```
array([3, 3, 3, 3, 3])
```

In [23]:
```
# Create a 1d array with 5 elements, filling them with the values 0 to 4
np.full(5, range(5))
```

Out[23]:
```
array([0, 1, 2, 3, 4])
```

In [24]:
```
# Sadly this trick won't work for multidimensional arrays
np.full([2, 5], range(10))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-ea647ac42aad> in <module>
      1 # Sadly this trick won't work for multidimensional arrays
----> 2 np.full([2, 5], range(10))

//anaconda3/lib/python3.7/site-packages/numpy/core/numeric.py in full(shape, fi
ll_value, dtype, order)
    334         dtype = array(fill_value).dtype
    335     a = empty(shape, dtype, order)
--> 336     multiarray.copyto(a, fill_value, casting='unsafe')
    337     return a
    338

ValueError: could not broadcast input array from shape (10) into shape (2,5)
```

In [25]:
```
# NumPy also has useful built-in mathematical numbers
np.full([2, 5], np.pi)
```

Out[25]:
```
array([[3.14159265, 3.14159265, 3.14159265, 3.14159265, 3.14159265],
       [3.14159265, 3.14159265, 3.14159265, 3.14159265, 3.14159265]])
```

Numpy array subsetting

You can subset NumPy arrays very similarly to list slicing in python.

In [18]:
```
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
print(x.shape)
x
```

```
(4, 3)
```

Out[18]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

In [19]:
```
# Retrieving the first row
x[0]
```

Out[19]:
```
array([1, 2, 3])
```

In [28]:
```
# Retrieving all rows after the first row
x[1:]
```

Out[28]:
```
array([[ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

This becomes particularly useful in multidimensional arrays when we can slice on multiple dimensions

In [29]:
```
# All rows, column 0
x[:,0]
```

Out[29]:
```
array([ 1,  4,  7, 10])
```

In [30]:
```
# Rows 2 through 4, columns 1 through 3
x[2:4,1:3]
```

Out[30]:
```
array([[ 8,  9],
       [11, 12]])
```

Notice that you can't slice in multiple dimensions naturally with built-in lists

In [1]:
```
x = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
x
```

Out[1]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

In [20]:
```
x[0]
```

Out[20]:
```
array([1, 2, 3])
```

In [21]:
```
x[:,0]
```

Out[21]:
```
array([ 1,  4,  7, 10])
```

In [34]:
```
# To slice along a second dimension with lists we must verbosely use a list compr
[i[0] for i in x]
```

Out[34]:
```
[1, 4, 7, 10]
```

In [23]:
```
# Doing this in multiple dimensions with lists
[i[1:3] for i in x[2:4]]
```

Out[23]:
```
[array([8, 9]), array([11, 12])]
```

3D Slicing

In [24]:
```
# With an array
x = np.array([
              [[1,2,3], [4,5,6]],
              [[7,8,9], [10,11,12]]
             ])
x
```

Out[24]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

In [25]:
```
x.shape
```

Out[25]:
```
(2, 2, 3)
```

In [26]:
```
x[:,:,-1]
```

Out[26]:
```
array([[ 3,  6],
       [ 9, 12]])
```

Summary

Great! You learned about a bunch of NumPy commands. Now, let's move over to the lab to put your new skills into practice!