Python Data Science Bite-Sized Lesson: Introduction to NumPy

Author: Michelle Franc Ragsac (mragsac@eng.ucsd.edu)

Notebook Information:

This Jupyter Notebook contains information on the basic functionality of the `numpy` package in Python for Module 4: Introduction to HPC.
It's running with a `Python 3` kernel!

Import Necessary Packages for the Module

Before we start coding, we want to import the `modules` that we'll be using in our notebook. This is the same as importing the modules at the beginning of a Python script.

For the three packages we'll be going through for this series of notebooks, they have different conventions for how they're called in people's code. The shorthand for `numpy` is `np`, `pandas` is `pd`, and `matplotlib.pyplot` is `plt`. If you find any code online (e.g., through StackOverflow) and you see these terms, these are the packages they're usually referring to!

In [1]:
```
import numpy as np                  # adds support for large, multi-dimensional arrays an
d optimized linear alg
import pandas as pd                 # adds support for Excel-like table operations (i.e.
 R Data Frames)
import matplotlib.pyplot as plt     # adds support for plotting in Python
```

In addition to importing these packages, there is a special line that we can add to view any plots generated with `matplotlib` within our notebook as part of the output of a code cell!

This special line is called a "magic function"!

Documentation on Magic Functions in Jupyter Notebooks: https://ipython.readthedocs.io/en/stable/interactive/magics.html (https://ipython.readthedocs.io/en/stable/interactive/magics.html)

In [2]:
```
%matplotlib inline
```

Even though we imported all of these packages, we'll only be going through `numpy` in this introductory notebook.

Introduction to NumPy

NumPy, which stands for "**Num**erical **Py**thon", is an open-source project that aims to enable numerical computing with Python. In Python, we have a data structure called `lists` that serve the purpose of arrays; however, they are slow to process. NumPy introduces a new type of array object within Python called the `ndarray` that is more efficient than the `list` structure. Additionally, NumPy also provides functions for working in the domain of linear algebra, matrices, and fourier transformations.

Note: This module adapts the content from `w3schools.com/python/numpy_intro.asp` (https://www.w3schools.com/python/numpy_intro.asp) into a Jupyter Notebook format to run through the code blocks.

NumPy Website: https://numpy.org/ (https://numpy.org/)
NumPy Code Base on GitHub: https://github.com/numpy/numpy (https://github.com/numpy/numpy)

Creating a Simple NumPy Array Called `arr`

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

print(f"The type of the variable arr is: {type(arr)}") # the type() function tells us th
e type of the object passed to it
print(f"Contents of arr: {arr}")
```

```
The type of the variable arr is: <class 'numpy.ndarray'>
Contents of arr: [1 2 3 4 5]
```

Creating NumPy Arrays with Varying Dimensions

An array can have any number of dimensions. We can manually create arrays with a certain number of dimentions by specifying particular values, or we can define the number of dimensions using the `ndmin` ("**N**umber of **D**i**M**ens**I**ons") argument.

```
In [4]: # Creating a 0-D array, or a scalar
a = np.array(10)

# Creating a 1-D array, or a uni-dimensional array
# these arrays are the most common and basic arrays
b = np.array([1, 2, 3, 4, 5])

# Creating a 2-D array, or a matrix
c = np.array([[1, 2, 3],
              [4, 5, 6]])

# Creating a 3-D array, or an array that has 2-D arrays as its elements
d = np.array([[[1, 2, 3], [4, 5, 6]],
              [[7, 8, 9], [10, 11, 12]]])

# Creating an array with 4 dimensions using the ndmin argument
e = np.array([1, 2, 3, 4, 5], ndmin=4)

# Checking the number of dimensions in each of our Numpy array objects
print(f"There are {a.ndim} dimensions in the Numpy Array called a")
print(f"There are {b.ndim} dimensions in the Numpy Array called b")
print(f"There are {c.ndim} dimensions in the Numpy Array called c")
print(f"There are {d.ndim} dimensions in the Numpy Array called d")
print(f"There are {e.ndim} dimensions in the Numpy Array called e")
```

```
There are 0 dimensions in the Numpy Array called a
There are 1 dimensions in the Numpy Array called b
There are 2 dimensions in the Numpy Array called c
There are 3 dimensions in the Numpy Array called d
There are 4 dimensions in the Numpy Array called e
```

The Shape of a NumPy Array

NumPy arrays have an attribute called `shape` that returns a tuple with each index having the number of corresponding elements at that dimension.

```
In [5]: # Print the shape of the 3-D array we created earlier called d
print(f"The shape of the array d is: {d.shape}")
```

```
The shape of the array d is: (2, 2, 3)
```

Accessing or Indexing NumPy Array Elements

Array indexing is the same as accessing an array element by referring to its index number.

Note: Within NumPy, indexes start at `0` instead of `1`!

For elements in 1-D arrays, we can simply specify the index that we want to access.

```
In [6]: # Access the third element in the 1-D array we created earlier called b
print(b[2])
```

```
3
```

To access elements from multidimensional arrays, we can use comma separated integers representing the dimensions and index of the element that we want to access.

```
In [7]: # Access one of the elements within the 3-D array we created earlier called d:
        print(d[0,1,2])

        # Within this command, we start off by accessing the first dimension of the array with i
        ndex 0
        # this gives us the two arrays: [[1, 2, 3], [4, 5, 6]]

        # Next, we access the first element of this array with index 1
        # this gives us the single array: [4, 5, 6]

        # Finally, we access the third element of this array with index 2
        # this gives us our final value: 6
```

```
6
```

Alternatively, we can use negative indexing the access an array from the end.

```
In [8]: # Let's access the second to last element from the 1-D array we created earlier called b
        print(b[-2])
```

```
4
```

Slicing NumPy Arrays

Within Python, slicing means taking the elements from a starting index to an ending index. For example, if we had the array `[1, 2, 3, 4, 5]`, we could **slice** this array to obtain the subarray, `[1, 2, 3]`.

```
In [9]: print(f"Slicing through elements in the array called b: {b}")

        # Slice elements 1 through 3 in the 1-D array we created earlier called b
        print(f"Slicing elements 1 to 3: {b[0:3]}")

        # Slice elements 3 through to the end of the 1-D array we created earlier called b
        print(f"Slicing elements 3 to the end: {b[3:]}")

        # Slice elements from the beginning to 4 of the 1-D array we created earlier called b
        print(f"Slicing elements from the start to 4: {b[:4]}")
```

```
Slicing through elements in the array called b: [1 2 3 4 5]
Slicing elements 1 to 3: [1 2 3]
Slicing elements 3 to the end: [4 5]
Slicing elements from the start to 4: [1 2 3 4]
```

Note: When we specify the end index when slicing, we include the start index, but not the end index!

In addition to positive slicing, we can also generate negative slices.

```
In [10]: # Slice through the element 3 from the end to the element 1 from the end
         print(f"Slicing elements 1 to 3: {b[-3:-1]}")
```

```
Slicing elements 1 to 3: [3 4]
```

Finally, we can also generate slices of multidimensional arrays!

We can do this similar to indexing the multidimensional array: we have to specify the portions of each dimension we want to slice out, with each dimension separated by a comma.

In [11]:
```
# Slice through a portion of the multidimensional array we previously generated called d
print(f"The original contents of d:\n{d}\n")
print(f"A slice taken from d:\n{d[0, 0:2, 0:1]}")
```

```
The original contents of d:
[[[ 1  2  3]
  [ 4  5  6]]

 [[ 7  8  9]
  [10 11 12]]]

A slice taken from d:
[[1]
 [4]]
```

Data Types in NumPy versus Default Types in Python

Within Python, we have five standard types: `string` , `integer` , `float` , `boolean` , and `complex` . In contrast to this, we have 11 additional data types in NumPy that are each represented by a single letter.

Data Type in NumPy	Single-Letter Code	Data Type in NumPy	Single-Letter Code
integer	`i`	boolean	`b`
unsigned integer	`u`	float	`f`
complex float	`c`	timedelta	`m`
datetime	`M`	object	`O`
string	`S`	unicode string	`U`
fixed chunk of memory for other type	`V`		

We can check the data type of an array by checking its `dtype` property.

In [12]:
```
# Let's check the dtype property of the multidimensional array we previously generated c
alled d
print(f"The type of d is: {d.dtype}")
```

```
The type of d is: int64
```

When we create NumPy arrays, we can also specify the type of the array using the `dtype` parameter!

In [13]:
```
# Create a new array called arr and designate the data type as a string
arr = np.array([1, 2, 3, 4, 5], dtype='S')

print(f"The contents of array arr are: {arr}")
print(f"The dtype of array arr is: {arr.dtype}")
```

```
The contents of array arr are: [b'1' b'2' b'3' b'4' b'5']
The dtype of array arr is: |S1
```

We can use the `astype()` method to convert the data type of an existing array. This method creates a copy of the array, and allows you to specify the data type as a paramter.

In [14]:
```python
# Create a new array called arr
arr = np.array([1.1, 2.2, 3.3, 4.4, 5.5])

print(f"The contents of array arr are: {arr}")
print(f"The dtype of array arr is: {arr.dtype}")

# Create a new array called arr_copy that is based on arr but with a new type
arr_copy = arr.astype('i')

print(f"\nThe contents of array arr_copy are: {arr_copy}")
print(f"The dtype of array arr_copy is: {arr_copy.dtype}")
```

```
The contents of array arr are: [1.1 2.2 3.3 4.4 5.5]
The dtype of array arr is: float64

The contents of array arr_copy are: [1 2 3 4 5]
The dtype of array arr_copy is: int32
```

Joining NumPy Arrays with `concatenate()`

We can join two or more NumPy arrays into a single array according to their axes.

To do this, all we have to do is pass a sequence of arrays we want to join to the `concatenate()` function, along with the axis we would like to join the arrays by. By default, the axis is taken as `0`.

In [15]:
```python
# Create two arrays we would like to manipulate in this section called first_half and se
cond_half
first_half = np.array([[1, 2, 3],[4, 5, 6]])
second_half = np.array([[7, 8, 9],[10, 11, 12]])

print(f"Created array #1:\n{first_half}")
print(f"Created array #2:\n{second_half}\n")

# Join the two arrays according to the default value, axis=0, then by the first axis, ax
is=1
print(f"Joining the two arrays according to axis=0:\n{np.concatenate((first_half, second
_half))}")              # joins along the columns
print(f"\nJoining the two arrays according to axis=1:\n{np.concatenate((first_half, seco
nd_half), axis=1)}")    # joins along the rows
```

```
Created array #1:
[[1 2 3]
 [4 5 6]]
Created array #2:
[[ 7  8  9]
 [10 11 12]]

Joining the two arrays according to axis=0:
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

Joining the two arrays according to axis=1:
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

Stacking NumPy Arrays as an Alternative to `concatenate()`

Stacking is an alternative way to concatenate arrays, however with stacking, we join the arrays along a new axis.

In [16]:
```
# Create two arrays we would like to manipulate in this section called arr1 and arr2
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

print(f"Created array #1: {arr1}")
print(f"Created array #2: {arr2}\n")

# Concatenate the two 1-D arrays along the second axis (axis=1) which results in putting
them one over the other
# (again, like with concatenate, the default axis is 0)

print(f"The result of stacking array #1 and array #2:\n{np.stack((arr1, arr2), axis=1)}"
)
```

```
Created array #1: [1 2 3]
Created array #2: [4 5 6]

The result of stacking array #1 and array #2:
[[1 4]
 [2 5]
 [3 6]]
```

Specifying the Direction of Stacking with `hstack()`, `vstack()`, and `dstack()`

To stack along the rows, we can use the method `hstack()`, or "horizontal stack".

In [17]:
```
print(f"The result of stacking array #1 and array #2 horizontally:\n{np.hstack((arr1, ar
r2))}")
```

```
The result of stacking array #1 and array #2 horizontally:
[1 2 3 4 5 6]
```

To stack along the columns, we can use the method `vstack()` or "vertical stack".

In [18]:
```
print(f"The result of stacking array #1 and array #2 vertically:\n{np.vstack((arr1, arr2
))}")
```

```
The result of stacking array #1 and array #2 vertically:
[[1 2 3]
 [4 5 6]]
```

Finally, we can stack along the height of the arrays (or depth), using the method `dstack()`, or "depth stack".

In [19]:
```
print(f"The result of stacking array #1 and array #2 vertically:\n{np.dstack((arr1, arr2
))}")
```

```
The result of stacking array #1 and array #2 vertically:
[[[1 4]
  [2 5]
  [3 6]]]
```

Splitting NumPy Arrays with `array_split()`

While joining NumPy arrays merges multiple arrays into one, splitting NumPy arrays breaks a single array into multiple.

To do this, we can use the `array_split()` method to specify the array we wish to split, along with the number of splits we would like to see. The return of this method is an array containing each split as an array; we can also index the resulting array.

In [20]:
```
# Create a new array called arr
arr = np.array([1, 2, 3, 4, 5, 6])

print(f"The contents of array arr are: {arr}")
print(f"The contents of array arr after splitting into 3 is: {np.array_split(arr, 3)}")
```

```
The contents of array arr are: [1 2 3 4 5 6]
The contents of array arr after splitting into 3 is: [array([1, 2]), array([3, 4]), arr
ay([5, 6])]
```

Note: If an array has less elements than required, then the method `array_split()` will adjust from the end accordingly. This is in contrast to the default method, `split()`, which is **not able to** adjust the elements when the elements are less in the source array.

Splitting NumPy arrays is not just limited to 1-D arrays. We can also split multi-dimensional arrays!

In [21]:
```
# Create a new 2-D array called arr
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])

print(f"The contents of array arr are:\n{arr}\n")
print(f"The contents of array arr are:\n{np.array_split(arr, 3)}")
```

```
The contents of array arr are:
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]

The contents of array arr are:
[array([[1, 2],
       [3, 4]]), array([[5, 6],
       [7, 8]]), array([[ 9, 10],
       [11, 12]])]
```

Note: Similar to the `stack()` method and its `hstack()`, `vstack()`, and `dstack()` variants, there are also `hsplit()`, `vsplit()`, and `dsplit()` variants that split along the rows, columns, and depth, respectively.

Searching NumPy Arrays with `where()`

We can search through arrays for a certain value of interest, and then return the indices that retrieve a match using the `where()` method.

In [22]:
```python
# Search through the array we generated before, arr, and search for where the value is 4
print(f"The contents of array arr are:\n{arr}\n")
print(f"The indices that match where an array element is 4 are: {np.where(arr == 4)}")
```

```
The contents of array arr are:
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]

The indices that match where an array element is 4 are: (array([1]), array([1]))
```

This result tell us that there is a match to a value of `4` at the position `(1, 1)` !

Note: In place of a single value, we can also replace the argument of `where()` with some sort of `boolean` expression, such as `where(arr%2 == 0)` , which is where any element within the array `arr` is even.

Sorting NumPy Arrays with `sort()`

We can use the `sort()` method to put the elements within a NumPy array into an ordered sequence (i.e., numerically or alphabetically sorted, ascending or descending in order).

Note: Sorting using the `sort()` method is not limited to numerical or alphabetical values: we can also sort `booleans` !

In [23]:
```python
# Create a new array called arr
arr = np.array([1, 3, 4, 6, 7, 5, 2])

print(f"The contents of array arr are: {arr}")
print(f"The contents of array arr after sorting are: {np.sort(arr)}")
```

```
The contents of array arr are: [1 3 4 6 7 5 2]
The contents of array arr after sorting are: [1 2 3 4 5 6 7]
```

Note: In the case of 2-D arrays, using the `sort()` method will sort both arrays.