MR2020: Coding for METOC

Module 4: Introduction to NumPy

What is NumPy?

"NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more."

(From the NumPy User Guide at https://numpy.org/doc/stable/user/whatisnumpy.html)

NumPy covers much of the functionality that MATLAB is used for in scientific computing, including in METOC research.

MATLAB vs NumPy: https://numpy.org/doc/stable/user/numpy-for-matlab-users.html

These slides are intended to introduce some basic capabilities of NumPy. Links are scattered throughout the slides that provide information on a plethora of additional capabilities.

NumPy Arrays

Arrays are the fundamental data structure in NumPy.

```
# Importing numpy
import numpy as np

# Create a 1D array.
arr1d = np.array([1,2,3])

# Create a 2D array.
arr2d = np.array([[1,2,3],[2,3,4]])

# Create a 3D array.
arr3D = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

Seldom will you create an array like this. More commonly, you might create placeholder arrays full of zeroes or ones or simply load existing data into an array.

Creating Placeholder Arrays

```
# Creating a 2D array full of zeros with dimensions 10 by 5.
arrzeros = np.zeros([10,5])

# Creating a 3D array full of ones with dimensions 3 by 3 by 10.
arrones = np.zeros([3,3,10])

# Create an array with a range of values given a start value, stop value, and step.
# Differs from Python range because decimal numbers may be used!
# In this case, create a range from 0 to 2 every 0.1.
arrarange = np.arange(0,2.1,0.1)

# Create arrays with a specified number of elements,
# and spaced equally between the specified beginning
# and end values
arrlinspace = np.linspace(0,2,5)
```

Operations with arrays

```
A = np.array([1,2,3])
B = np.array([2,3,4])

# Add
A + B # Returns array([3,5,7])

# Subtract
A - B # Returns array([-1,-1,-1])

# Multiply element-wise (VERY different from MATLAB!)
A * B # Returns array([2,6,12])

# Divide element-wise (also VERY different than MATLAB!)
A / B # Returns array([0.5,0.6666667,0.75])

# Exponent
A ** B # Returns array([1,8,81])
```

Arrays must have the same dimensions to perform operations on them together.

What happens when you add two Python lists instead of NumPy arrays?

Order of Operations

```
A = np.array([1,2,3])
B = np.array([2,3,4])

# Example 1
(A + B)**(A*B)
# array([ 9, 15625, 13841287201])


# Example 2
A + B**(A*B)
# array([ 5, 731, 16777219])


# Example 3
(A+B)**A*B
# array([ 6, 75, 1372])
```

1. Parentheses
2. Exponents
3. Products and Quotients
4. Additions and Subtractions

Commonly Used Functions

```
# Sum an array
np.sum(A)

# Mean of an array
np.mean(A)

# Median of an array
np.median(A)

# Maximum value in array
np.max(A)

# Minimum value in array
np.min(A)
```

```
# Cosine
np.cos(A)

# Sine
np.sin(A)

# Tangent
np.tan(A)

# Convert degrees to radians
np.deg2rad(A)

# Combine them
np.cos(np.deg2rad(A))
```

For more math functions (and ChatGPT will know these):
https://numpy.org/doc/stable/reference/routines.math.html

Handling Missing Data

METOC datasets often contain missing data. Maybe you're looking at satellite data and there was a temporary problem with scanning that causes data to not be collected. Or maybe you are looking at a time series of sea surface temperature data from a buoy and the instrument malfunctioned for a couple of days before it was repaired. How do you handle missing data (which is very common in METOC applications) without ruining your analysis?

Suppose you have an array like this:

```
A = np.array([2,3,np.nan,4])
```

What happens if you do `np.mean(A)`? You get nan (not a number).
NumPy has special functions to deal with this.

```
# Max of NaN-containing array.
np.nanmax(A)
# Min of NaN-containing array.
np.nanmin(A)
# Mean of NaN-containing array.
np.nanmean(A)
# Median of NaN-containing array.
np.nanmedian(A)
```

Linear Algebra

The fundamental data construct in MATLAB (Matrix Laboratory) is the matrix. In contrast, NumPy uses the array as its basic construct. However, NumPy is still capable of easily completing matrix operations. SciPy has some redundant and additional capabilities.

```
A = np.array([1,2,3])
B = np.array([2,3,4])

# Dot product
dotprod = np.dot(A,B)

# Matrix multiplication
matprod = np.matmul(A,B)
```

More info here: https://numpy.org/doc/stable/reference/routines.linalg.html

Checking Dimensions, Shape, Size of Array

```
A = np.array([[1,2,3],[2,3,4]])

# Size
# Returns integer representing total number of elements in array
np.size(A)
A.size

# Shape
# Returns tuple containing number of rows, then number of columns
np.shape(A)
A.shape

# Number of dimensions
# Returns integer representing the number of dimensions in array
np.ndim(A)
A.ndim
```

} These two lines do the same thing.

```
A = np.array([[1,2,3],[2,3,4]])
```

This line calls the function from NumPy and applies to it A. This works but is not necessary because A is a NumPy array object already. However, if A were a list of lists such as

```
A = [[1,2,3],[2,3,4]]
```

then only np.size(A) would work and A.size would not.

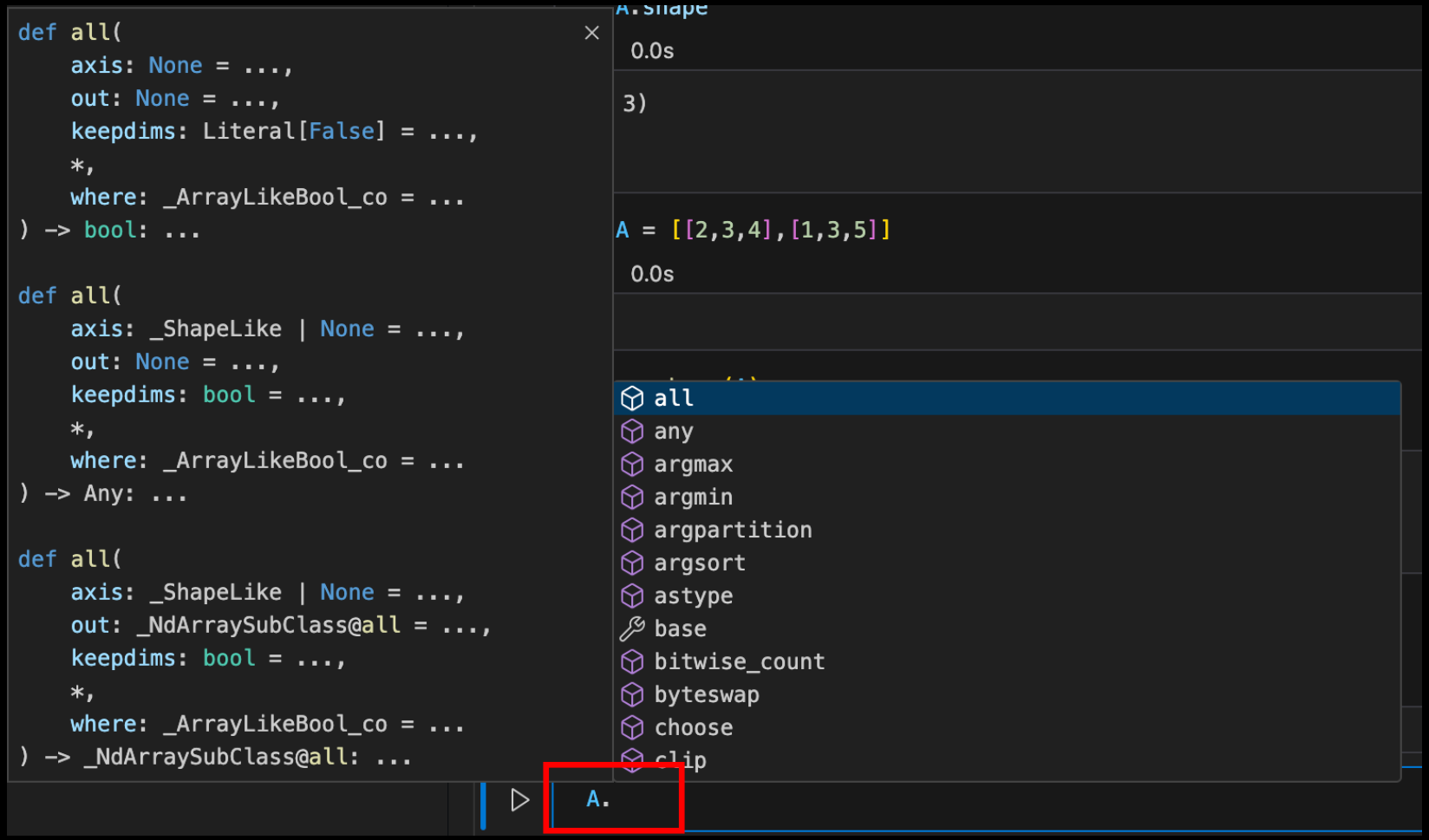
```
np.size(A)
```

For A as assigned above, these two lines do the same thing.

```
A.size
```

This line also returns the size of A, but because A is already a NumPy object, it has several attributes assigned to it. One of those is its size. So, the object A has an attribute size (among others). Getting an attribute requires putting a period after the variable name then coding the attribute after it.

Hint: In the VSC interactive window, entering the variable name and a period will a scrollable list of functions and attributes that can be applied to that variable.



Reshaping Arrays

Sometimes you may need to change the shape of an array. There are many reasons this may happen. Perhaps you have an array that needs to be transposed before plotting. Perhaps you need to speed up an operation and you need to make your array one-dimensional. Below are some common NumPy methods for reshaping arrays:

```
A = np.random.randint(10,size=(4,4))
```

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

These are rows.

These are columns.

```
# Transpose array
A.transpose()
```

```
# Reshape the 4 x 4 array to a 2 x 8 array.
A.reshape(2,8)
```

```
# Flatten the array to 1D.
A.flatten()
```

Transposing

This is A

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

```
A.transpose()
```

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

Transposing an array swaps its rows and columns.

In this case, the shape remains 4 x 4 because A was square. But if A were for example, 3 by 6, it would become 6 by 3.

Reshaping

This is A

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

4 * 4 = 16

You can only reshape an array so that it has the same size as the original.

2 rows

8 columns

```
A.reshape(2,8)
```

2 * 8 = 16

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

In this case, the reshape array has fewer rows than the original.
What would happen if we reshaped to 8 x 2 instead?

Flattening

This is A

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

Flattening an array reduces it to 1 dimension by stacking the rows together one after another into a single row.

```
A.flatten()
```

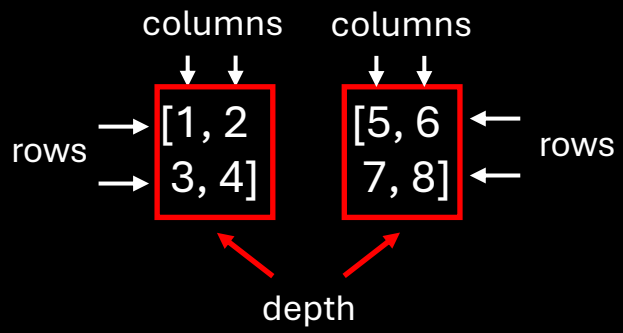
```
array([3, 3, 3, 6, 8, 5, 5, 4, 2, 3, 4, 1, 1, 5, 4, 4])
```

Stacking and Concatenating Arrays

NumPy arrays can also be combined with each other, in a process Python calls stacking. Below are some examples for different ways to stack two 3D arrays.

```
array1 = np.array([
[[1, 2], [3, 4]],
[[5, 6], [7, 8]]
])
```

```
array2 = np.array([
[[9, 10], [11, 12]],
[[13, 14], [15, 16]]
])
```

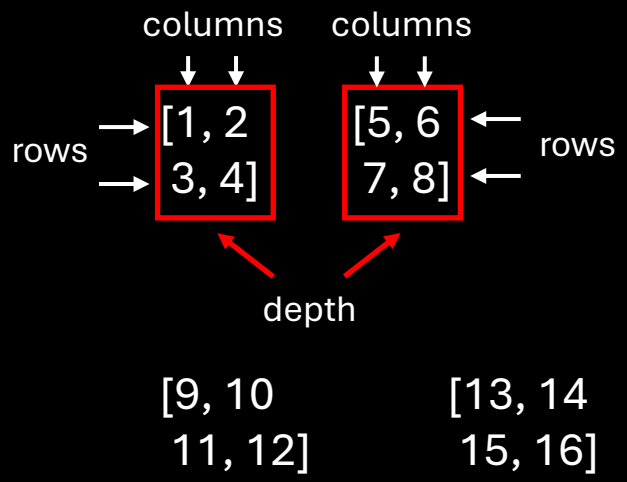


Suppose we want to “stick” the two arrays together. How can we do this?

Stacking and Concatenating Arrays

```
array1 = np.array([
[[1, 2], [3, 4]],
[[5, 6], [7, 8]]
])
```

```
array2 = np.array([
[[9, 10], [11, 12]],
[[13, 14], [15, 16]]
])
```



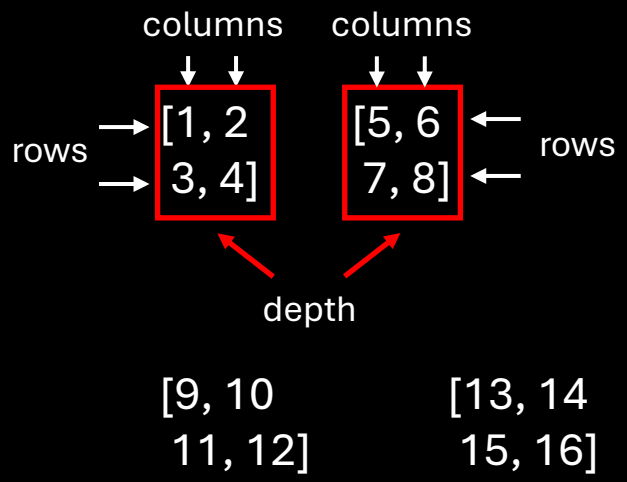
Stack arrays along first dimension, which in this case will make the resulting array deeper.

```
np.vstack((array1,array2))
```

Stacking and Concatenating Arrays

```
array1 = np.array([
[[1, 2], [3, 4]],
[[5, 6], [7, 8]]
])
```

```
array2 = np.array([
[[9, 10], [11, 12]],
[[13, 14], [15, 16]]
])
```



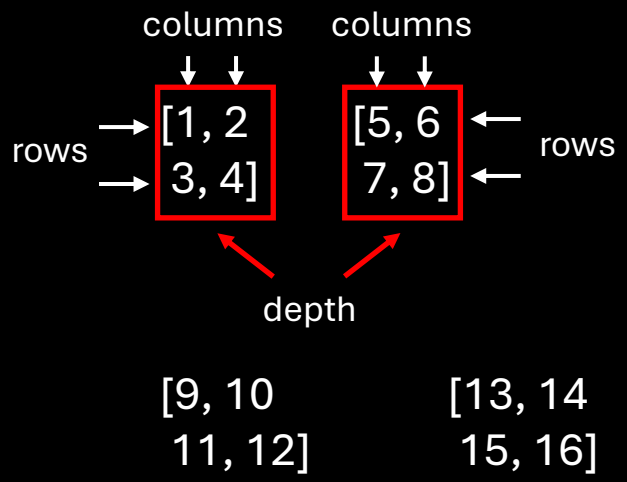
Stack arrays along second dimension, which in this case will make the resulting array have more rows.

```
np.hstack((array1,array2))
```

Stacking and Concatenating Arrays

```
array1 = np.array([
[[1, 2], [3, 4]],
[[5, 6], [7, 8]]
])
```

```
array2 = np.array([
[[9, 10], [11, 12]],
[[13, 14], [15, 16]]
])
```



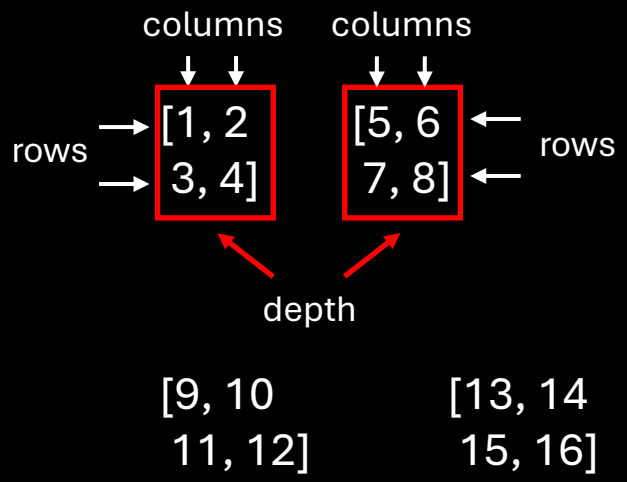
Stack arrays along third dimension, which in this case will make the resulting array have more columns. For 2D arrays, a third dimension will be added to the resulting stacked array.

```
np.dstack((array1,array2))
```

Stacking and Concatenating Arrays

```
array1 = np.array([
[[1, 2], [3, 4]],
[[5, 6], [7, 8]]
])
```

```
array2 = np.array([
[[9, 10], [11, 12]],
[[13, 14], [15, 16]]
])
```



Make the different arrays a new dimension themselves, creating a new array with, in this case, a fourth dimension, that would be indexed in the position indicated by axis. Axis = 0 means the new dimension will be indexed first.

```
np.stack((array1,array2),axis=0)
```