NumPy_Notes

February 3, 2022

1 NumPy Notes

1.0.1 Author: Jesse Randall

1.0.2 Date: 20/01/2022

1.1 Preface

This notebook is meant for people who are familiar with the base functionality (syntax, methods, libraries, etc.) of Python and are looking to learn or review the contents of the NumPy library.

1.1.1 Changelog

25/01/2022: Always look at the documentation! 25/01/2022: Always look at the documentation!

I used string concatenation in the print statements to make one input from multiple strings because using multiple comma separated inputs resulted in misaligned statements for the output.

Example: Misaligned output with comma separated inputs.

Input: print('Hello. My name is Jesse. \n', 'The output of this line is NOT aligned with the output of the first line.')

Output: Hello. My name is Jesse. The output of this line is NOT aligned with the output of the first line.

Example: Aligned output with concatenated inputs.

Input: print('Hello. My name is Jesse. \n' + 'The output of this line IS aligned with the output of the first line.')

Output: Hello. My name is Jesse. The output of this line IS aligned with the output of the first line.

This was a workaround I found through some light experimentation. However, I found that my original method of string concatenation using the '+' operator is of O(N^2) complexity in Python, where N is the number of characters in each string, because strings are immutable meaning they cannot be changed. Every time string concatenation is performed using the '+' operator, a new string is made resulting in the minimum O(N^2) complexity if all strings are of the same size.

I replaced string concatenation using the '+' operator with the str.join() method which takes a list of the strings that are going to be concatenated. This method creates exactly one string to contain the list of strings provided meaning it is O(N) complexity which is significantly more efficient.

Again this was just a workaround where I did not bother to understand why the print() function has this behavior. What I should have done was look at the documentation for the print() function. I finally did look at the documentation and found a keyword argument called 'sep' that defines the character used to separate multiple inputs to the print() function. By default it is a single space and this is why the outputs were misaligned. I changed it to an empty string and the problem went away WITHOUT the need of string concatenation which simplifies the complexity of the code by removing an unnecessary step.

Example: Aligned output using multiple inputs with sep=''.

Input: »> print('Hello. My name is Jesse. \n', 'The output of this line IS aligned with the output of the first line.', sep='')

Output: Hello. My name is Jesse. The output of this line IS aligned with the output of the first line.

This is meant to be a beginner guide to NumPy and I do not want to demonstrate inefficient coding practice. Newer programmers will use this as an example of what 'good' coding looks like so I want to make sure that I demonstrate efficient coding practice throughout.

02/02/2022: Introducing Jupyter Notebook Created a section at the beginning to teach people how to use a Jupyter Notebook.

1.2 How to use a Jupyter Notebook

This is a Jupyter Notebook detailing how to use the NumPy package for Python. This section covers how to use a Jupyter Notebook for people who are not familiar.

Each selected "cell" or block of text/code is run through an interactive Python (iPython) kernel when you use the keyboard command "Shit+Enter". You can select a cell by clicking inside the box containing the text/code or by clicking to the left side of the cell. A cell is selected when is has a rectangular bar to the left of the cell, it is colored blue by default.

Cells that contain code to be run will have the closed brackets on the left of the cell that look like this "[ ]". A cell has been run when you see a number inside the corresponding brackets for the cell and the number corresponds to how many cells of code have been run so far. Re-running a cell or running a new cell will increment the number. There will usually be a printed output beneath a cell that contains code. Try running the simple print statement below using the "Shift+Enter" command when you have the cell selected.

[1]:
```
print('Hello, World!')
```

```
Hello, World!
```

You can go through each cell individually with the "Shift+Enter" method of running a cell. Make sure you start from the beginning by selecting the first cell in the sequence of code, otherwise the code will not work and there will be printed errors in the output signifying that. The notebook will automatically select the next cell after running the selected cell so you can easily run each cell in the sequence by repeatedly using the "Shift+Enter" command. To run all cells at once, click on the "Run" tab in the top left and click the option "Run All Cells". All cells in the entire notebook will be run and you can scroll through the notebook to see the output for each cell containing code.

You can reset the current Kernel by selecting the "Kernel" tab in the top left and selecting the "Restart Kernel and Clear All Outputs" option. This will remove all currently defined functions and variables effectively resetting the notebook back to its base state.

The notebook may have already been run when you receive it. This is easily determined by looking at the brackets beside each cell. If there is a number inside each input cell with code, then the notebook has been fully executed and the associated output for each cell should be printed.

1.3 Import NumPy

[2]:
```
# Import NumPy using abreviation 'np' for easier typing. Standard shorthand. Also
# import other useful libraries that will be used.

import numpy as np # NumPy library. Used to instantiate arrays and perform array␣
 ↪operations.
import traceback   # Traceback library. Used to print the call stack when an␣
 ↪error occurs.

# This statement shortens the printed float numbers for easier reading.
np.set_printoptions(formatter={'float': lambda x: "{0:0.3f}".format(x)})

# Initializizng variables for NumPy array dimensions and general use.
N1 = 5         # Size of NumPy array dimension 1.
N2 = int(1e1) # Size of NumPy array dimension 2.
N3 = int(1e1) # Size of NumPy array dimension 3.
NL = '\n'      # New line or return. Makes coding print statements simpler.
LS = 2         # Number of 'newline' or 'return' commands used when printing.
```

1.4 Initialization of NumPy Arrays

Initialization and properties of NumPy arrays. Example np.random.random_sample. Standard function for initiating an array of random values in the interval [0,1). Takes a tuple to define the shape or size of each dimension for the array.

[3]:
```
# Initialization. Different intialization for different array types. Check␣
 ↪documentation.
a = np.random.random_sample( (N1,N2,N3) )
print('The NumPy array \"a\": \n' , str(a) , NL , sep='')
```

```
The NumPy array "a":
[[[0.345 0.037 0.804 0.180 0.355 0.121 0.544 0.936 0.830 0.916]
  [0.077 0.964 0.317 0.793 0.491 0.776 0.450 0.322 0.482 0.826]
  [0.548 0.907 0.787 0.338 0.654 0.017 0.726 0.563 0.694 0.095]
  [0.081 0.273 0.865 0.577 0.544 0.418 0.156 0.914 0.764 0.910]
  [0.344 0.359 0.568 0.763 0.963 0.766 0.901 0.263 0.031 0.880]
  [0.149 0.973 0.474 0.482 0.826 0.154 0.103 0.600 0.594 0.796]
  [0.848 0.317 0.417 0.617 0.499 0.840 0.636 0.218 0.675 0.985]
  [0.415 0.568 0.672 0.240 0.364 0.158 0.547 0.442 0.707 0.122]
```

```
 [0.846 0.096 0.712 0.450 0.105 0.236 0.035 0.112 0.743 0.082]
 [0.338 0.213 0.936 0.136 0.617 0.773 0.030 0.831 0.427 0.583]]

[[0.759 0.780 0.940 0.786 0.785 0.063 0.719 0.611 0.343 0.651]
 [0.712 0.492 0.416 0.086 0.849 0.327 0.066 0.365 0.022 0.750]
 [0.107 0.421 0.121 0.058 0.831 0.829 0.042 0.849 0.712 0.198]
 [0.840 0.781 0.938 0.564 0.625 0.324 0.827 0.787 0.860 0.730]
 [0.259 0.710 0.250 0.885 0.557 0.314 0.434 0.789 0.147 0.203]
 [0.776 0.795 0.491 0.279 0.455 0.537 0.230 0.855 0.504 0.319]
 [0.416 0.647 0.398 0.319 0.569 0.768 0.138 0.461 0.331 0.681]
 [0.564 0.070 0.512 0.182 0.592 0.052 0.993 0.516 0.995 0.872]
 [0.653 0.422 0.850 0.171 0.190 0.770 0.996 0.209 0.882 0.686]
 [0.989 0.340 0.134 0.363 0.761 0.596 0.396 0.510 0.159 0.384]]

[[0.194 0.480 0.143 0.614 0.996 0.011 0.800 0.427 0.652 0.871]
 [0.852 0.406 0.760 0.745 0.712 0.143 0.711 0.789 0.490 0.275]
 [0.508 0.841 0.169 0.201 0.138 0.495 0.858 0.928 0.525 0.146]
 [0.613 0.772 0.064 0.697 0.832 0.707 0.930 0.447 0.528 0.721]
 [0.111 0.393 0.741 0.519 0.644 0.505 0.031 0.093 0.969 0.622]
 [0.105 0.077 0.085 0.132 0.055 0.800 0.751 0.371 0.961 0.206]
 [0.892 0.177 0.055 0.777 0.340 0.715 0.759 0.263 0.657 0.873]
 [0.323 0.379 0.006 0.024 0.530 0.088 0.717 0.410 0.835 0.558]
 [0.160 0.275 0.937 0.188 0.622 0.110 0.465 0.794 0.744 0.958]
 [0.630 0.478 0.732 0.204 0.454 0.705 0.974 0.048 0.303 0.239]]

[[0.270 0.796 0.746 0.709 0.550 0.159 0.551 0.131 0.353 0.475]
 [0.379 0.415 0.407 0.877 0.134 0.306 0.788 0.754 0.458 0.055]
 [0.350 0.970 0.823 0.809 0.488 0.317 0.010 0.320 0.341 0.224]
 [0.504 0.109 0.359 0.955 0.031 0.233 0.635 0.586 0.098 0.364]
 [0.764 0.504 0.530 0.916 0.979 0.778 0.438 0.536 0.182 0.612]
 [0.346 0.862 0.821 0.791 0.630 0.290 0.741 0.106 0.898 0.586]
 [0.552 0.653 0.783 0.037 0.621 0.237 0.830 0.638 0.678 0.061]
 [0.653 0.620 0.562 0.927 0.940 0.728 0.239 0.395 0.741 0.076]
 [0.865 0.020 0.367 0.233 0.606 0.316 0.397 0.904 0.720 0.825]
 [0.919 0.322 0.786 0.839 0.469 0.545 0.221 0.787 0.230 0.603]]

[[0.180 0.465 0.905 0.664 0.639 0.687 0.003 0.731 0.289 0.464]
 [0.919 0.053 0.714 0.743 0.907 0.095 0.743 0.974 0.328 0.576]
 [0.583 0.824 0.839 0.093 0.328 0.611 0.146 0.571 0.531 0.913]
 [0.324 0.985 0.413 0.430 0.750 0.596 0.125 0.440 0.142 0.391]
 [0.940 0.908 0.980 0.576 0.428 0.599 0.955 0.920 0.126 0.137]
 [0.261 0.334 0.657 0.228 0.823 0.860 0.237 0.716 0.738 0.121]
 [0.047 0.186 0.207 0.625 0.533 0.909 0.726 0.249 0.787 0.978]
 [0.887 0.693 0.884 0.042 0.094 0.171 0.689 0.162 0.306 0.350]
 [0.049 0.227 0.740 0.784 0.749 0.033 0.855 0.083 0.741 0.928]
 [0.525 0.541 0.919 0.015 0.387 0.174 0.927 0.654 0.271 0.732]]]
```

[4]:
```
# Properties of NumPy array "a".
print('The data type for all elements: '                , str(a.dtype)    , NL ,
      'The size of each element in memory in bytes: '   , str(a.itemsize) , NL ,
      'The total number of elements: '                  , str(a.size)     , NL ,
      'The total number of dimensions: '                , str(a.ndim)     , NL ,
      'The shape or number of elements per dimension: ' , str(a.shape)    , NL ,
      'The maximum value of contained elements: '       , str(a.max())    , NL ,
      'The minimum value of contained elements: '       , str(a.min())    , NL ,␣
 ↪sep='')
```

```
The data type for all elements: float64
The size of each element in memory in bytes: 8
The total number of elements: 500
The total number of dimensions: 3
The shape or number of elements per dimension: (5, 10, 10)
The maximum value of contained elements: 0.9959737977452803
The minimum value of contained elements: 0.0028117880961360253
```

1.5 Indexing/Slicing to access elements

Indexing and Slicing to access elements. Uses "0 indexing" meaning first element of each dimension is referenced with number "0". Negative numbers start at the end of each dimension. The notation [start:stop:step] is known as a "slice" and can be used for granular control when indexing a specific axis.

Reference: NumPy Documentation v1.22 URL: https://numpy.org/doc/stable/user/basics.indexing.html Quote: "All arrays generated by basic slicing are always views of the original array." Comment: A "view" is an array whose elements point to the corresponding elements in the indexed array. This is relevant later on when we learn about boolean array masking where we are manipulating the values of an array.

[5]:
```
# Get a specific element using a[d1,d2,...,dn] notation.
print('First element(s): ' , str((a[0,0,0]   , a[1,0,0]   , a[2,0,0]))   , NL ,
      'Last element(s):  ' , str((a[0,-1,-1], a[1,-1,-1], a[2,-1,-1])) , NL ,␣
 ↪sep='')
```

```
First element(s): (0.3451339077231168, 0.7587127651752653, 0.19407314287004906)
Last element(s):  (0.5826849265377851, 0.38422577518608825, 0.23925693640750267)
```

[6]:
```
# Get a specific 'row'. ':' used to select all elements of specified dimension.
print('First row(s): \n' , str(a[0,0,:])  , NL*LS ,
      'Last row(s):  \n' , str(a[0,-1,:]) , NL , sep='')
```

```
First row(s):
[0.345 0.037 0.804 0.180 0.355 0.121 0.544 0.936 0.830 0.916]

Last row(s):
```

```
[0.338 0.213 0.936 0.136 0.617 0.773 0.030 0.831 0.427 0.583]
```

[7]:
```
# Get a specific 'column'.
print('First column(s): \n' , str(a[0,:,0])  , NL*LS ,
      'Last column(s):  \n' , str(a[0,:,-1]) , NL , sep='')
```

```
First column(s):
[0.345 0.077 0.548 0.081 0.344 0.149 0.848 0.415 0.846 0.338]

Last column(s):
[0.916 0.826 0.095 0.910 0.880 0.796 0.985 0.122 0.082 0.583]
```

[8]:
```
# Get a specific 'table'.
print('First table(s): \n' , str(a[0,:,:])  , NL*LS ,
      'Last table(s):  \n' , str(a[-1,:,:]) , NL , sep='')
```

```
First table(s):
[[0.345 0.037 0.804 0.180 0.355 0.121 0.544 0.936 0.830 0.916]
 [0.077 0.964 0.317 0.793 0.491 0.776 0.450 0.322 0.482 0.826]
 [0.548 0.907 0.787 0.338 0.654 0.017 0.726 0.563 0.694 0.095]
 [0.081 0.273 0.865 0.577 0.544 0.418 0.156 0.914 0.764 0.910]
 [0.344 0.359 0.568 0.763 0.963 0.766 0.901 0.263 0.031 0.880]
 [0.149 0.973 0.474 0.482 0.826 0.154 0.103 0.600 0.594 0.796]
 [0.848 0.317 0.417 0.617 0.499 0.840 0.636 0.218 0.675 0.985]
 [0.415 0.568 0.672 0.240 0.364 0.158 0.547 0.442 0.707 0.122]
 [0.846 0.096 0.712 0.450 0.105 0.236 0.035 0.112 0.743 0.082]
 [0.338 0.213 0.936 0.136 0.617 0.773 0.030 0.831 0.427 0.583]]

Last table(s):
[[0.180 0.465 0.905 0.664 0.639 0.687 0.003 0.731 0.289 0.464]
 [0.919 0.053 0.714 0.743 0.907 0.095 0.743 0.974 0.328 0.576]
 [0.583 0.824 0.839 0.093 0.328 0.611 0.146 0.571 0.531 0.913]
 [0.324 0.985 0.413 0.430 0.750 0.596 0.125 0.440 0.142 0.391]
 [0.940 0.908 0.980 0.576 0.428 0.599 0.955 0.920 0.126 0.137]
 [0.261 0.334 0.657 0.228 0.823 0.860 0.237 0.716 0.738 0.121]
 [0.047 0.186 0.207 0.625 0.533 0.909 0.726 0.249 0.787 0.978]
 [0.887 0.693 0.884 0.042 0.094 0.171 0.689 0.162 0.306 0.350]
 [0.049 0.227 0.740 0.784 0.749 0.033 0.855 0.083 0.741 0.928]
 [0.525 0.541 0.919 0.015 0.387 0.174 0.927 0.654 0.271 0.732]]
```

[9]:
```
# Granular indexing or 'slicing' using
# a[d1=startindex:endindex:stepsize,d2,...,dn] notation.
print('First row, every other element: \n' , str(a[0,0,0:-1:2])  , NL*LS ,
      'Last row, every other element:  \n' , str(a[0,-1,0:-1:2]) , NL , sep='')
```

```
First row, every other element:
```

```
[0.345 0.804 0.355 0.544 0.830]

Last row, every other element:
[0.338 0.936 0.617 0.030 0.427]
```

1.6 More Initialization Examples

1.6.1 Basic Initialization

[10]:
```
# More array initialization examples.

# All elements initiated as zeros.
print('All elements initiated as zeros: \n' ,
      str(np.zeros((N1,N1), dtype='int32')) , NL , sep='')
```

```
All elements initiated as zeros:
[[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]]
```

[11]:
```
# All elements initiated as ones.
print('All elements initiated as ones: \n' ,
      str(np.ones((N1,N1), dtype='int32')) , NL , sep='')
```

```
All elements initiated as ones:
[[1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]]
```

[12]:
```
# All elements initiated as a specified value.
print('All elements initiated as a specified value: \n' ,
      str(np.full((N1,N1), N2, dtype='float32')) , NL , sep='')
```

```
All elements initiated as a specified value:
[[10.000 10.000 10.000 10.000 10.000]
 [10.000 10.000 10.000 10.000 10.000]
 [10.000 10.000 10.000 10.000 10.000]
 [10.000 10.000 10.000 10.000 10.000]
 [10.000 10.000 10.000 10.000 10.000]]
```

[13]:
```
# All elements initiated as a letter.
print('All elements initiated as a letter: \n' ,
      str(np.full((N1,N1), 'a')) , NL , sep='')
```

```
All elements initiated as a letter:
[['a' 'a' 'a' 'a' 'a']
 ['a' 'a' 'a' 'a' 'a']
 ['a' 'a' 'a' 'a' 'a']
 ['a' 'a' 'a' 'a' 'a']
 ['a' 'a' 'a' 'a' 'a']]
```

[14]:
```
# Initialize an array using an exisiting array as an example.
print('All elements initiated as a \"full like\" matrix: \n' ,
      str(np.full_like(a[0],N1)) , NL , sep='')
```

```
All elements initiated as a "full like" matrix:
[[5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000]
 [5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000]
 [5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000]
 [5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000]
 [5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000]
 [5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000]
 [5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000]
 [5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000]
 [5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000]
 [5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000 5.000]]
```

[15]:
```
# 2D Identity Matrix.
print('Initiated 2D Identity Matrix: \n' , str(np.identity(N1)) , NL , sep='')
```

```
Initiated 2D Identity Matrix:
[[1.000 0.000 0.000 0.000 0.000]
 [0.000 1.000 0.000 0.000 0.000]
 [0.000 0.000 1.000 0.000 0.000]
 [0.000 0.000 0.000 1.000 0.000]
 [0.000 0.000 0.000 0.000 1.000]]
```

Repeating array. This is tricky due to the indexing required for proper initialization. The second index for the array 'a' must be written as '0:1' so that it returns a 2D array with shape (1,N3). This allows specification of the axis that we want repeated as shown in the first example where 'axis=0' is specified. If the indexing is written as '0' then a 1D array is returned with shape '(N3,)' and does not allow row repetition, only element repetition as shown in the following statement.

[16]:
```
print('First row of array \"a\" repeated N1 times: \n' ,
      str(np.repeat(a[0,0:1,:],N1,axis=0)) , NL*LS ,
      'First row elements of array \"a\" repeated N1 times: \n' ,
```

```
      str(np.repeat(a[0,0,:],N1))          , NL , sep='')
```

```
First row of array "a" repeated N1 times:
[[0.345 0.037 0.804 0.180 0.355 0.121 0.544 0.936 0.830 0.916]
 [0.345 0.037 0.804 0.180 0.355 0.121 0.544 0.936 0.830 0.916]
 [0.345 0.037 0.804 0.180 0.355 0.121 0.544 0.936 0.830 0.916]
 [0.345 0.037 0.804 0.180 0.355 0.121 0.544 0.936 0.830 0.916]
 [0.345 0.037 0.804 0.180 0.355 0.121 0.544 0.936 0.830 0.916]]

First row elements of array "a" repeated N1 times:
[0.345 0.345 0.345 0.345 0.345 0.037 0.037 0.037 0.037 0.037 0.804 0.804
 0.804 0.804 0.804 0.180 0.180 0.180 0.180 0.180 0.355 0.355 0.355 0.355
 0.355 0.121 0.121 0.121 0.121 0.121 0.544 0.544 0.544 0.544 0.544 0.936
 0.936 0.936 0.936 0.936 0.830 0.830 0.830 0.830 0.830 0.916 0.916 0.916
 0.916 0.916]
```

1.6.2 Special Initialization

Lets put our knowledge of array initialization to the test.

We are going to create the following array: [[1.,1.,1.,1.,1.] [1.,0.,0.,0.,1.] [1.,0.,9.,0.,1.] [1.,0.,0.,0.,1.] [1.,1.,1.,1.,1.]] This will scale to any shape as long as both dimensions have odd size.

[17]:
```
# Start with array full of ones.
testShape = (N1,N1)
testIndex = np.ones(testShape, dtype='int32')
print('Array \"testIndex\": \n' , str(testIndex) , NL , sep='')
```

```
Array "testIndex":
[[1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]]
```

[18]:
```
# Fill the inside with zeros.
testIndex[1:testShape[0]-1,1:testShape[1]-1] = np.
 ↪zeros((testShape[0]-2,testShape[1]-2), dtype='float32')
print('Modified \"testIndex\": \n' , str(testIndex) , NL , sep='')
```

```
Modified "testIndex":
[[1 1 1 1 1]
 [1 0 0 0 1]
 [1 0 0 0 1]
 [1 0 0 0 1]
 [1 1 1 1 1]]
```

[19]:
```
# Assign the middle value to 9 and we're done.
testIndex[testShape[0]//2,testShape[1]//2] = 9.
print('Final \"testArrray\": \n' , str(testIndex) , NL , sep='')
```

```
Final "testArrray":
[[1 1 1 1 1]
 [1 0 0 0 1]
 [1 0 9 0 1]
 [1 0 0 0 1]
 [1 1 1 1 1]]
```

1.7 Assignment is NOT copying

This section shows the important distinction between assignment and proper initialization of a NumPy array or any variable for that matter. You may be altering the contents of an array when you do not want to and being conscientious about when you are using assignment vs array initialization is the key to preventing this type of error.

[20]:
```
# Lets start by initializing an array as testA and assigning it to a different
# variable called testB.

testA = np.zeros((N1,N1), dtype='int32')
testB = testA

print('Array \"testA\": \n' , str(testA) , NL*LS ,
      'Array \"testB\": \n' , str(testB) , NL , sep='')
```

```
Array "testA":
[[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]]

Array "testB":
[[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]]
```

[21]:
```
# Alright. We now have two variables that point to an array. Lets try changing
# the values in the array that testB is assigned to.
testB[0,:] = N2
print('First Test: Change value in \"testB\" and check result.' , NL*LS ,
      'Array \"testA\": \n' , str(testA) , NL*LS ,
```

```
      'Array \"testB\": \n' , str(testB) , NL , sep='')
```

```
First Test: Change value in "testB" and check result.

Array "testA":
[[10 10 10 10 10]
 [ 0  0  0  0  0]
 [ 0  0  0  0  0]
 [ 0  0  0  0  0]
 [ 0  0  0  0  0]]

Array "testB":
[[10 10 10 10 10]
 [ 0  0  0  0  0]
 [ 0  0  0  0  0]
 [ 0  0  0  0  0]
 [ 0  0  0  0  0]]
```

Oh. That is NOT good. Changing the values in testB is the same as changing values in testA because they both point to the exact same array in memory. We 'assigned' testB to the array that was 'initialized' when defining testA. If we want to make a proper 'copy' of the array that testA is assigned to then we need to 'initialize' a new array using a method of numpy arrays called 'copy'.

[22]:
```
# Initialize new arrays and test the difference.
testA = np.zeros((N1,N1), dtype='int32')
testB = testA.copy()

# Now we have two initialized arrays that are identical in value but stored in
# separate memory locations. Changing one does not change the other.
testB[0,:] = N2
print('Second Test: Change value in testB and check result.' , NL*LS ,
      'Array testA: \n' , str(testA) , NL*LS ,
      'Array testB: \n' , str(testB) , NL , sep='')
```

```
Second Test: Change value in testB and check result.

Array testA:
[[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]]

Array testB:
[[10 10 10 10 10]
 [ 0  0  0  0  0]
 [ 0  0  0  0  0]
 [ 0  0  0  0  0]
```

```
 [ 0  0  0  0  0]]
```

1.8 Advanced Indexing

NumPy arrays can be indexed using NumPy arrays or any list/tuple/etc. Wild stuff. We can create useful NumPy arrays for indexing using boolean operators on an array that determines whether each element in an array satisfies a given condition. It initiates a boolean 'mask' that is an array with the same shape as the original array and contains either a "True" (1) or "False" (0) value for every element denoting whether the corresponding element from the original array satisfied the given condition.

[23]:
```
# Find every element in the array 'a' that has a value greater than a.max()/2
# This statement initiates a boolean mask array according to the provided␣
 ↪condition.
aMaskHalfMax = (a > (a.max()/2))
print('aMaskHalfMax.shape: ' , str(aMaskHalfMax.shape) , NL ,
      'aMaskHalfMax.size:  ' , str(aMaskHalfMax.size)  , NL ,
      'aMaskHalfMax: \n'     , str(aMaskHalfMax)       , NL*LS , sep='')
```

```
aMaskHalfMax.shape: (5, 10, 10)
aMaskHalfMax.size:  500
aMaskHalfMax:
[[[False False  True False False False  True  True  True  True]
  [False  True False  True False  True False False False  True]
  [ True  True  True False  True False  True  True  True False]
  [False False  True  True  True False False  True  True  True]
  [False False  True  True  True  True  True False False  True]
  [False  True False False  True False False  True  True  True]
  [ True False False  True  True  True  True False  True  True]
  [False  True  True False False False  True False  True False]
  [ True False  True False False False False False  True False]
  [False False  True False  True  True False  True False  True]]

 [[ True  True  True  True  True False  True  True False  True]
  [ True False False False  True False False False False  True]
  [False False False False  True  True False  True  True False]
  [ True  True  True  True  True False  True  True  True  True]
  [False  True False  True  True False False  True False False]
  [ True  True False False False  True False  True  True False]
  [False  True False False  True  True False False False  True]
  [ True False  True False  True False  True  True  True  True]
  [ True False  True False False  True  True False  True  True]
  [ True False False False  True  True False  True False False]]

 [[False False False  True  True False  True False  True  True]
  [ True False  True  True  True False  True  True False False]
  [ True  True False False False False  True  True  True False]
```

```
 [ True  True False  True  True  True  True False  True  True]
 [False False  True  True  True  True False False  True  True]
 [False False False False False  True  True False  True False]
 [ True False False  True False  True  True False  True  True]
 [False False False False  True False  True False  True  True]
 [False False  True False  True False False  True  True  True]
 [ True False  True False False  True  True False False False]]

[[False  True  True  True  True False  True False False False]
 [False False False  True False False  True  True False False]
 [False  True  True  True False False False False False False]
 [ True False False  True False False  True  True False False]
 [ True  True  True  True  True  True False  True False  True]
 [False  True  True  True  True False  True False  True  True]
 [ True  True  True False  True False  True  True  True False]
 [ True  True  True  True  True  True False False  True False]
 [ True False False False  True False False  True  True  True]
 [ True False  True  True False  True False  True False  True]]

[[False False  True  True  True  True False  True False False]
 [ True False  True  True  True False  True  True False  True]
 [ True  True  True False False  True False  True  True  True]
 [False  True False False  True  True False False False False]
 [ True  True  True  True False  True  True  True False False]
 [False False  True False  True  True False  True  True False]
 [False False False  True  True  True  True False  True  True]
 [ True  True  True False False False  True False False False]
 [False False  True  True  True False  True False  True  True]
 [ True  True  True False False False  True  True False  True]]
```

[24]:
```
# Index array "a" using the created mask.
aIndexMaskHalfMax = a[aMaskHalfMax]
print('aIndexMaskHalfMax.shape: ' , str(aIndexMaskHalfMax.shape) , NL ,
      'aIndexMaskHalfMax.size:  ' , str(aIndexMaskHalfMax.size)  , NL ,
      'aIndexMaskHalfMax: \n'     , str(aIndexMaskHalfMax)       , NL , sep='')
```

```
aIndexMaskHalfMax.shape: (269,)
aIndexMaskHalfMax.size:  269
aIndexMaskHalfMax:
[0.804 0.544 0.936 0.830 0.916 0.964 0.793 0.776 0.826 0.548 0.907 0.787
 0.654 0.726 0.563 0.694 0.865 0.577 0.544 0.914 0.764 0.910 0.568 0.763
 0.963 0.766 0.901 0.880 0.973 0.826 0.600 0.594 0.796 0.848 0.617 0.499
 0.840 0.636 0.675 0.985 0.568 0.672 0.547 0.707 0.846 0.712 0.743 0.936
 0.617 0.773 0.831 0.583 0.759 0.780 0.940 0.786 0.785 0.719 0.611 0.651
 0.712 0.849 0.750 0.831 0.829 0.849 0.712 0.840 0.781 0.938 0.564 0.625
 0.827 0.787 0.860 0.730 0.710 0.885 0.557 0.789 0.776 0.795 0.537 0.855
```

```
0.504 0.647 0.569 0.768 0.681 0.564 0.512 0.592 0.993 0.516 0.995 0.872
0.653 0.850 0.770 0.996 0.882 0.686 0.989 0.761 0.596 0.510 0.614 0.996
0.800 0.652 0.871 0.852 0.760 0.745 0.712 0.711 0.789 0.508 0.841 0.858
0.928 0.525 0.613 0.772 0.697 0.832 0.707 0.930 0.528 0.721 0.741 0.519
0.644 0.505 0.969 0.622 0.800 0.751 0.961 0.892 0.777 0.715 0.759 0.657
0.873 0.530 0.717 0.835 0.558 0.937 0.622 0.794 0.744 0.958 0.630 0.732
0.705 0.974 0.796 0.746 0.709 0.550 0.551 0.877 0.788 0.754 0.970 0.823
0.809 0.504 0.955 0.635 0.586 0.764 0.504 0.530 0.916 0.979 0.778 0.536
0.612 0.862 0.821 0.791 0.630 0.741 0.898 0.586 0.552 0.653 0.783 0.621
0.830 0.638 0.678 0.653 0.620 0.562 0.927 0.940 0.728 0.741 0.865 0.606
0.904 0.720 0.825 0.919 0.786 0.839 0.545 0.787 0.603 0.905 0.664 0.639
0.687 0.731 0.919 0.714 0.743 0.907 0.743 0.974 0.576 0.583 0.824 0.839
0.611 0.571 0.531 0.913 0.985 0.750 0.596 0.940 0.908 0.980 0.576 0.599
0.955 0.920 0.657 0.823 0.860 0.716 0.738 0.625 0.533 0.909 0.726 0.787
0.978 0.887 0.693 0.884 0.689 0.740 0.784 0.749 0.855 0.741 0.928 0.525
0.541 0.919 0.927 0.654 0.732]
```

[25]:
```
# Create a copy of array "a" so that we can alter the contents.
aTest = a.copy()

# Turn the specified values to zero.
aTest[aMaskHalfMax] = 0
print('Turn the specified values to zero. \n', str(aTest) , NL , sep='')
```

```
Turn the specified values to zero.
[[[0.345 0.037 0.000 0.180 0.355 0.121 0.000 0.000 0.000 0.000]
  [0.077 0.000 0.317 0.000 0.491 0.000 0.450 0.322 0.482 0.000]
  [0.000 0.000 0.000 0.338 0.000 0.017 0.000 0.000 0.000 0.095]
  [0.081 0.273 0.000 0.000 0.000 0.418 0.156 0.000 0.000 0.000]
  [0.344 0.359 0.000 0.000 0.000 0.000 0.000 0.263 0.031 0.000]
  [0.149 0.000 0.474 0.482 0.000 0.154 0.103 0.000 0.000 0.000]
  [0.000 0.317 0.417 0.000 0.000 0.000 0.000 0.218 0.000 0.000]
  [0.415 0.000 0.000 0.240 0.364 0.158 0.000 0.442 0.000 0.122]
  [0.000 0.096 0.000 0.450 0.105 0.236 0.035 0.112 0.000 0.082]
  [0.338 0.213 0.000 0.136 0.000 0.000 0.030 0.000 0.427 0.000]]

 [[0.000 0.000 0.000 0.000 0.000 0.063 0.000 0.000 0.343 0.000]
  [0.000 0.492 0.416 0.086 0.000 0.327 0.066 0.365 0.022 0.000]
  [0.107 0.421 0.121 0.058 0.000 0.000 0.042 0.000 0.000 0.198]
  [0.000 0.000 0.000 0.000 0.000 0.324 0.000 0.000 0.000 0.000]
  [0.259 0.000 0.250 0.000 0.000 0.314 0.434 0.000 0.147 0.203]
  [0.000 0.000 0.491 0.279 0.455 0.000 0.230 0.000 0.000 0.319]
  [0.416 0.000 0.398 0.319 0.000 0.000 0.138 0.461 0.331 0.000]
  [0.000 0.070 0.000 0.182 0.000 0.052 0.000 0.000 0.000 0.000]
  [0.000 0.422 0.000 0.171 0.190 0.000 0.000 0.209 0.000 0.000]
  [0.000 0.340 0.134 0.363 0.000 0.000 0.396 0.000 0.159 0.384]]
```

```
[[0.194 0.480 0.143 0.000 0.000 0.011 0.000 0.427 0.000 0.000]
 [0.000 0.406 0.000 0.000 0.000 0.143 0.000 0.000 0.490 0.275]
 [0.000 0.000 0.169 0.201 0.138 0.495 0.000 0.000 0.000 0.146]
 [0.000 0.000 0.064 0.000 0.000 0.000 0.000 0.447 0.000 0.000]
 [0.111 0.393 0.000 0.000 0.000 0.000 0.031 0.093 0.000 0.000]
 [0.105 0.077 0.085 0.132 0.055 0.000 0.000 0.371 0.000 0.206]
 [0.000 0.177 0.055 0.000 0.340 0.000 0.000 0.263 0.000 0.000]
 [0.323 0.379 0.006 0.024 0.000 0.088 0.000 0.410 0.000 0.000]
 [0.160 0.275 0.000 0.188 0.000 0.110 0.465 0.000 0.000 0.000]
 [0.000 0.478 0.000 0.204 0.454 0.000 0.000 0.048 0.303 0.239]]

[[0.270 0.000 0.000 0.000 0.000 0.159 0.000 0.131 0.353 0.475]
 [0.379 0.415 0.407 0.000 0.134 0.306 0.000 0.000 0.458 0.055]
 [0.350 0.000 0.000 0.000 0.488 0.317 0.010 0.320 0.341 0.224]
 [0.000 0.109 0.359 0.000 0.031 0.233 0.000 0.000 0.098 0.364]
 [0.000 0.000 0.000 0.000 0.000 0.000 0.438 0.000 0.182 0.000]
 [0.346 0.000 0.000 0.000 0.000 0.290 0.000 0.106 0.000 0.000]
 [0.000 0.000 0.000 0.037 0.000 0.237 0.000 0.000 0.000 0.061]
 [0.000 0.000 0.000 0.000 0.000 0.000 0.239 0.395 0.000 0.076]
 [0.000 0.020 0.367 0.233 0.000 0.316 0.397 0.000 0.000 0.000]
 [0.000 0.322 0.000 0.000 0.469 0.000 0.221 0.000 0.230 0.000]]

[[0.180 0.465 0.000 0.000 0.000 0.000 0.003 0.000 0.289 0.464]
 [0.000 0.053 0.000 0.000 0.000 0.095 0.000 0.000 0.328 0.000]
 [0.000 0.000 0.000 0.093 0.328 0.000 0.146 0.000 0.000 0.000]
 [0.324 0.000 0.413 0.430 0.000 0.000 0.125 0.440 0.142 0.391]
 [0.000 0.000 0.000 0.000 0.428 0.000 0.000 0.000 0.126 0.137]
 [0.261 0.334 0.000 0.228 0.000 0.000 0.237 0.000 0.000 0.121]
 [0.047 0.186 0.207 0.000 0.000 0.000 0.000 0.249 0.000 0.000]
 [0.000 0.000 0.000 0.042 0.094 0.171 0.000 0.162 0.306 0.350]
 [0.049 0.227 0.000 0.000 0.000 0.033 0.000 0.083 0.000 0.000]
 [0.000 0.000 0.000 0.015 0.387 0.174 0.000 0.000 0.271 0.000]]]
```

[26]:
```
# Turn the specified values back to their original value.
aTest[aMaskHalfMax] = aIndexMaskHalfMax
print('Turn the specified values back to their original value. \n' , str(aTest)␣
 ↪, NL , sep='')
```

```
Turn the specified values back to their original value.
[[[0.345 0.037 0.804 0.180 0.355 0.121 0.544 0.936 0.830 0.916]
  [0.077 0.964 0.317 0.793 0.491 0.776 0.450 0.322 0.482 0.826]
  [0.548 0.907 0.787 0.338 0.654 0.017 0.726 0.563 0.694 0.095]
  [0.081 0.273 0.865 0.577 0.544 0.418 0.156 0.914 0.764 0.910]
  [0.344 0.359 0.568 0.763 0.963 0.766 0.901 0.263 0.031 0.880]
  [0.149 0.973 0.474 0.482 0.826 0.154 0.103 0.600 0.594 0.796]
  [0.848 0.317 0.417 0.617 0.499 0.840 0.636 0.218 0.675 0.985]
  [0.415 0.568 0.672 0.240 0.364 0.158 0.547 0.442 0.707 0.122]
```

```
 [0.846 0.096 0.712 0.450 0.105 0.236 0.035 0.112 0.743 0.082]
 [0.338 0.213 0.936 0.136 0.617 0.773 0.030 0.831 0.427 0.583]]

[[0.759 0.780 0.940 0.786 0.785 0.063 0.719 0.611 0.343 0.651]
 [0.712 0.492 0.416 0.086 0.849 0.327 0.066 0.365 0.022 0.750]
 [0.107 0.421 0.121 0.058 0.831 0.829 0.042 0.849 0.712 0.198]
 [0.840 0.781 0.938 0.564 0.625 0.324 0.827 0.787 0.860 0.730]
 [0.259 0.710 0.250 0.885 0.557 0.314 0.434 0.789 0.147 0.203]
 [0.776 0.795 0.491 0.279 0.455 0.537 0.230 0.855 0.504 0.319]
 [0.416 0.647 0.398 0.319 0.569 0.768 0.138 0.461 0.331 0.681]
 [0.564 0.070 0.512 0.182 0.592 0.052 0.993 0.516 0.995 0.872]
 [0.653 0.422 0.850 0.171 0.190 0.770 0.996 0.209 0.882 0.686]
 [0.989 0.340 0.134 0.363 0.761 0.596 0.396 0.510 0.159 0.384]]

[[0.194 0.480 0.143 0.614 0.996 0.011 0.800 0.427 0.652 0.871]
 [0.852 0.406 0.760 0.745 0.712 0.143 0.711 0.789 0.490 0.275]
 [0.508 0.841 0.169 0.201 0.138 0.495 0.858 0.928 0.525 0.146]
 [0.613 0.772 0.064 0.697 0.832 0.707 0.930 0.447 0.528 0.721]
 [0.111 0.393 0.741 0.519 0.644 0.505 0.031 0.093 0.969 0.622]
 [0.105 0.077 0.085 0.132 0.055 0.800 0.751 0.371 0.961 0.206]
 [0.892 0.177 0.055 0.777 0.340 0.715 0.759 0.263 0.657 0.873]
 [0.323 0.379 0.006 0.024 0.530 0.088 0.717 0.410 0.835 0.558]
 [0.160 0.275 0.937 0.188 0.622 0.110 0.465 0.794 0.744 0.958]
 [0.630 0.478 0.732 0.204 0.454 0.705 0.974 0.048 0.303 0.239]]

[[0.270 0.796 0.746 0.709 0.550 0.159 0.551 0.131 0.353 0.475]
 [0.379 0.415 0.407 0.877 0.134 0.306 0.788 0.754 0.458 0.055]
 [0.350 0.970 0.823 0.809 0.488 0.317 0.010 0.320 0.341 0.224]
 [0.504 0.109 0.359 0.955 0.031 0.233 0.635 0.586 0.098 0.364]
 [0.764 0.504 0.530 0.916 0.979 0.778 0.438 0.536 0.182 0.612]
 [0.346 0.862 0.821 0.791 0.630 0.290 0.741 0.106 0.898 0.586]
 [0.552 0.653 0.783 0.037 0.621 0.237 0.830 0.638 0.678 0.061]
 [0.653 0.620 0.562 0.927 0.940 0.728 0.239 0.395 0.741 0.076]
 [0.865 0.020 0.367 0.233 0.606 0.316 0.397 0.904 0.720 0.825]
 [0.919 0.322 0.786 0.839 0.469 0.545 0.221 0.787 0.230 0.603]]

[[0.180 0.465 0.905 0.664 0.639 0.687 0.003 0.731 0.289 0.464]
 [0.919 0.053 0.714 0.743 0.907 0.095 0.743 0.974 0.328 0.576]
 [0.583 0.824 0.839 0.093 0.328 0.611 0.146 0.571 0.531 0.913]
 [0.324 0.985 0.413 0.430 0.750 0.596 0.125 0.440 0.142 0.391]
 [0.940 0.908 0.980 0.576 0.428 0.599 0.955 0.920 0.126 0.137]
 [0.261 0.334 0.657 0.228 0.823 0.860 0.237 0.716 0.738 0.121]
 [0.047 0.186 0.207 0.625 0.533 0.909 0.726 0.249 0.787 0.978]
 [0.887 0.693 0.884 0.042 0.094 0.171 0.689 0.162 0.306 0.350]
 [0.049 0.227 0.740 0.784 0.749 0.033 0.855 0.083 0.741 0.928]
 [0.525 0.541 0.919 0.015 0.387 0.174 0.927 0.654 0.271 0.732]]]
```

[27]:
```
# The function np.any() will find whether a given condition is satisfied by ANY
# elements along the given axes.

condition1  = (a == a.max())
aAnyHalfMax = np.any(condition1, axis=(1,2))
print('Do ANY elements of the tables in array \"a\" satisfy the given condition?
 ↪' , NL ,
      'aAnyHalfMax: ' , str(aAnyHalfMax) , NL ,
      'This tells us that table ' , np.where(aAnyHalfMax == True)[0] ,
      ' in array \"a\" contains the element with the maximum value.' , sep='')
```

```
Do ANY elements of the tables in array "a" satisfy the given condition?
aAnyHalfMax: [False  True False False False]
This tells us that table [1] in array "a" contains the element with the maximum
value.
```

[28]:
```
# The function np.all() will find whether a given condition is satisfied by ALL
# elements along the given axes.

# Create boolean mask array using the condition we are testing.
# Then pass it to the function np.all and specify the axes.
condition2   = (a < a.max())
aAllLessMax = np.all(condition2, axis=(1,2))

print('Do ALL elements of the tables in array \"a\" satisfy the given condition?
 ↪' , NL ,
      'aAllInterval: ' , str(aAllLessMax) , NL ,
      'This tells us that table ' , np.where(aAllLessMax == False)[0] ,
      ' contains the element in array \"a\" with the maximum value.' , sep='')
```

```
Do ALL elements of the tables in array "a" satisfy the given condition?
aAllInterval: [ True False  True  True  True]
This tells us that table [1] contains the element in array "a" with the maximum
value.
```

1.9 Mathematical Operations on NumPy arrays

Now we will look at the mathematical operations you can perform over the elements of a NumPy array. Arithmetic operations between a scalar and a NumPy array are generally 'elementwise' meaning they are applied to every element of the array individually. We do not need to loop over the array to apply the operation to all of its elements. This process is automated by the NumPy array. This is the reason we use NumPy arrays for data analysis as it simplifies the code and is more efficient in terms of its runtime/complexity.

1.9.1 Arithmetic operations between scalar and NumPy array

[29]:
```
# Initialize array for operations.
b = np.full((N1,N1), N1 , dtype='int32')
print('Array b= \n' , str(b) , NL , sep='')
```

```
Array b=
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
```

[30]:
```
# Addition
print('Array b+' , str(N1) , ' = \n' , str(b+N1) , NL , sep='')
```

```
Array b+5 =
[[10 10 10 10 10]
 [10 10 10 10 10]
 [10 10 10 10 10]
 [10 10 10 10 10]
 [10 10 10 10 10]]
```

[31]:
```
# Subtraction
print('Array b-' , str(N1) , ' = \n' , str(b-N1) , NL , sep='')
```

```
Array b-5 =
[[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]]
```

[32]:
```
# Multiplication
print('Array b*' , str(N1) , ' = \n' , str(b*N1) , NL , sep='')
```

```
Array b*5 =
[[25 25 25 25 25]
 [25 25 25 25 25]
 [25 25 25 25 25]
 [25 25 25 25 25]
 [25 25 25 25 25]]
```

[33]:
```
# Division
print('Array b/' , str(N1) , ' = \n' , str(b/N1) , NL , sep='')
```

```
Array b/5 =
[[1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]]
```

[34]:
```
# Exponentiation (Respresented using the '**' symbol)
print('Array b^2 = \n' , str(b**2) , NL , sep='')
```

```
Array b^2 =
[[25 25 25 25 25]
 [25 25 25 25 25]
 [25 25 25 25 25]
 [25 25 25 25 25]
 [25 25 25 25 25]]
```

1.9.2 Functions applied to NumPy Arrays

[35]:
```
# You can also apply functions.
print('Array sin(b) = \n'               , str(np.sin(b)) , NL*LS ,
      'Array cos(b) = \n'               , str(np.cos(b)) , NL*LS ,
      'Array exp(b) = \n'               , str(np.exp(b)) , NL*LS ,
      'Array sin(b)^2 + cos(b)^2 = \n' , str(np.sin(b)**2 + np.cos(b)**2) ,␣
 ↪sep='')
```

```
Array sin(b) =
[[-0.959 -0.959 -0.959 -0.959 -0.959]
 [-0.959 -0.959 -0.959 -0.959 -0.959]
 [-0.959 -0.959 -0.959 -0.959 -0.959]
 [-0.959 -0.959 -0.959 -0.959 -0.959]
 [-0.959 -0.959 -0.959 -0.959 -0.959]]

Array cos(b) =
[[0.284 0.284 0.284 0.284 0.284]
 [0.284 0.284 0.284 0.284 0.284]
 [0.284 0.284 0.284 0.284 0.284]
 [0.284 0.284 0.284 0.284 0.284]
 [0.284 0.284 0.284 0.284 0.284]]

Array exp(b) =
[[148.413 148.413 148.413 148.413 148.413]
 [148.413 148.413 148.413 148.413 148.413]
 [148.413 148.413 148.413 148.413 148.413]
 [148.413 148.413 148.413 148.413 148.413]
 [148.413 148.413 148.413 148.413 148.413]]
```

```
Array sin(b)^2 + cos(b)^2 =
[[1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]]
```

1.9.3 Operations between NumPy arrays

[36]:
```
# What happens when we try to do arithmetic between two arrays?
# Initialize array c as a copy of b.
c = b.copy()
print('Array "b"= \n' , str(b) , NL ,
      'Array "c"= \n' , str(c) , sep='')
```

```
Array "b"=
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
Array "c"=
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
```

[37]:
```
# Addition
print('Array b+c= \n' , str(b) , NL , '+' , NL , str(c) , NL ,
      '=' , NL , str(b+c) , NL , sep='')
```

```
Array b+c=
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
+
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
=
[[10 10 10 10 10]
 [10 10 10 10 10]
```

```
 [10 10 10 10 10]
 [10 10 10 10 10]
 [10 10 10 10 10]]
```

[38]:
```
# Subtraction
print('Array b-c= \n' , str(b) , NL , '-' , NL , str(c) , NL ,
      '=' , NL , str(b-c) , NL , sep='')
```

```
Array b-c=
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
-
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
=
[[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]]
```

[39]:
```
# Multiplication
print('Array b*c= \n' , str(b) , NL , '*' , NL , str(c) , NL ,
      '=' , NL , str(b*c) , NL , sep='')
```

```
Array b*c=
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
*
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
=
[[25 25 25 25 25]
 [25 25 25 25 25]
 [25 25 25 25 25]
```

```
 [25 25 25 25 25]
 [25 25 25 25 25]]
```

[40]:
```
# Division
print('Array b/c= \n' , str(b) , NL , '/' , NL , str(c) , NL ,
      '=' , NL , str(b/c) , NL , sep='')
```

```
Array b/c=
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
/
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
=
[[1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]
 [1.000 1.000 1.000 1.000 1.000]]
```

Ok. It applied element by element arithmetic. Very useful to know. What happens when we try to do arithmetic operations between two arrays that do not have the same shape?

[41]:
```
# This "try except" sequence allows execution of code that produces an error.
# The statement underneath "except:" prints the last error in the stack trace.
try:
    print('Shape of array a: ' , str(a.shape) , NL ,
          'Shape of array b: ' , str(b.shape) , NL , sep='')
    print('Array a+b= \n'       , str(a+b)     , NL)
except:
    traceback.print_exc()
```

```
Shape of array a: (5, 10, 10)
Shape of array b: (5, 5)


Traceback (most recent call last):
  File "C:\Users\spotn\AppData\Local\Temp/ipykernel_15032/2001332600.py", line
6, in <module>
    print('Array a+b= \n'       , str(a+b)     , NL)
ValueError: operands could not be broadcast together with shapes (5,10,10) (5,5)
```

Ah. It did not like that. "Operands could not be broadcast together with shapes (N1,N2,N3) (N1,N1)". This implies that the operations between arrays is known as "broadcasting" and that the shape of the arrays is important when performing operations that utilize broadcasting. Broadcasting is the most important feature of NumPy arrays as we will find out in the next section.

1.10 Broadcasting

Source: NumPy Documentation v1.22 (All quotes regarding broadcasting) URL: https://numpy.org/doc/stable/user/basics.broadcasting.html

Quote: "Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations. There are, however, cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computation."

Comment: So, broadcasting defines the behavior of operations on arrays. Lets try to learn a little more about broadcasting through further experimentation and practice.

[42]:
```
# Initialize array for broadcast testing
d = np.ones((N1), dtype='int32')
print('Now we have array \"b\" with shape ' , str(b.shape) , NL , str(b) , NL ,
      'And array \"d\" with shape '          , str(d.shape) , NL , str(d) , NL ,␣
 ↪sep='')
```

```
Now we have array "b" with shape (5, 5)
[[5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]
 [5 5 5 5 5]]
And array "d" with shape (5,)
[1 1 1 1 1]
```

[43]:
```
# What happens if we try to add these arrays together?
print('Array b+d = \n'       , str(b+d)          , NL ,
      'Array (b+d).shape: ' , str((b+d).shape) , NL , sep='')
```

```
Array b+d =
[[6 6 6 6 6]
 [6 6 6 6 6]
 [6 6 6 6 6]
 [6 6 6 6 6]
 [6 6 6 6 6]]
Array (b+d).shape: (5, 5)
```

Ok that worked. This, along with what we learned inthe last seciton, should allow us to infer what is happening when we add b+d. Let's look at the documentation to double check our understand-

ing.

Quote: "NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape."

Comment: This is what we saw in the section on operations between NumPy arrays where both arrays "b" and "c" had the same shape. This type of broad-casting is necessary when you have two arrays with unique values that cannot be represented by a single variable.

[44]:
```
# Example: Broadcasting between arrays with same shape.
print('b.shape = '      , str(b.shape)    , NL ,
      'c.shape = '      , str(c.shape)    , NL ,
      '(b+c).shape = ' , str((b+c).shape) , sep='')
```

```
b.shape = (5, 5)
c.shape = (5, 5)
(b+c).shape = (5, 5)
```

Quote: "NumPy's broadcasting rule relaxes this constraint when the arrays' shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation. We can think of the scalar being stretched during the arithmetic operation into an array with the same shape as the NumPy array being operated on."

Comment: This is what we observed in the section detailing arithmetic operations between a scalar and a NumPy array. This method of broadcasting is moreefficient because less memory is moved around during the operation. It is more memory intensive to work with an array containing elements that all have the same value than a scalar with the same value.

[45]:
```
# Example: Broadcasting between scalar and an array.
print('(N1*b).shape =' , str((N1*b).shape))
```

```
(N1*b).shape = (5, 5)
```

Quote: "When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when

1. they are equal

2. one of them is one

If these conditions are not met, a ValueError: operands could not be broadcast together exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs."

Comment: This is what we observed in this section between array "b" and array "d". This is very important to understand when trying to perform operations between arrays of different sizes. NumPy broadcasting is not magic and can only be used in certain situations.

[46]:
```
# Example: Initialize arrays and look at the shape of the resulting array after␣
 ↪broadcasting.
bcA = np.ones((8,1,6,1)) # (4d array):  8 x 1 x 6 x 1
bcB = np.ones(  (7,1,5)) # (3d array):      7 x 1 x 5
```

```
bcC = bcA + bcB           # (4d array):  8 x 7 x 6 x 5
print('bcC.shape =' , bcC.shape)
```

```
bcC.shape = (8, 7, 6, 5)
```

1.11 Comments and Thoughts

This is just a preview of what NumPy arrays have to offer. These properties make NumPy arrays an essential tool for people that requires easy and efficient manipulation of large data structures. I personally have used NumPy arrays in all of my scientific projects that require the manipulation of large, digital data structures because of their versatility.

Remember that NumPy arrays are not always the best tool for the job and using them requires careful consideration of the pros and cons when compared to something else.

Pros:

Cons: