NumPy library

Before starting...

What is a library?

A collection of related modules (written python code files `.py`) that you can use instead of writing from scratch. Libraries help save time and effort by providing reusable functions for different purposes.

How do you import libraries to your Jupyter Notebook?

- To install library in your laptop, check installation guide from respective libary.
  - In this case, we will check NumPy's documentation: https://numpy.org/install/
  - In Jupter Notebook, you can use `!pip install numpy`
- To import libaries so you can use functions, use `import <libary_name> as <alias>`, where alias is a short name asigned to the library
- It is convention to import NumPy using the code: `import numpy as np`
- To only import the array part of NumPy use the code: `from numpy import array`

In [104…
```
!pip install numpy
```

```
Requirement already satisfied: numpy in c:\programdata\anaconda3\lib\site-packages (1.24.3)
```

In [105…
```
import numpy as np
```

Here is a list of some libraries you might have to install for the course. Check the documentation of each library:

- Pandas => https://pandas.pydata.org/docs/getting_started/install.html
- Matplotlib => https://matplotlib.org/stable/install/index.html
- Seaborn => https://seaborn.pydata.org/installing.html

NumPy arrays

- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50 times faster than traditional Python lists.
- Numpy arrays should all have the same data type (usually numeric).
- The mathematical operations that are meant to be performed on arrays would be extremely inefficient if the arrays weren’t homogeneous (https://numpy.org/doc/stable/user/absolute_beginners.html)
- Use the np.array function to create a numpy array: `arr = np.array([1, 2, 3, 4, 5])`
- **Notice** the `library_name.function` call style. This will be used a lot.

In [106…
```
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

```
[1 2 3 4 5]
```

In [ ]:
```
# help(np.array)
```

In [108…
```
print("arr variable is type:", type(arr))
```

```
arr variable is type: <class 'numpy.ndarray'>
```

- You can create an array of 0's:

In [109…
```
np.zeros(8)
```

Out[109]:
```
array([0., 0., 0., 0., 0., 0., 0., 0.])
```

- You can easily create a range of elements in a numpy array:

In [110…
```
np.arange(1,6,2)
```
Out[110]:
```
array([1, 3, 5])
```

- What is the code in the next two cells doing?

In [111…
```
np.arange(3, 57, 4)
```
Out[111]:
```
array([ 3,  7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55])
```

In [ ]:
```
# help(np.linspace)
```

In [113…
```
np.linspace(0, 20, num=9)
```
Out[113]:
```
array([ 0. ,  2.5,  5. ,  7.5, 10. , 12.5, 15. , 17.5, 20. ])
```

NumPy arrays sorting, filtering and indexing

- `height = np.array([172, 159, 175, 180, 167, 173])` is a one-dimensional numpy array.
- We can index values similarly to lists eg `height[0]`.
- `height[0:3]` gives the first three entries of height in a numpy array: `([172, 159, 175])`

In [114…
```
height = np.array([172, 159, 175, 180, 167, 173])
```

In [115…
```
# print first element in array
# zero-indexed

print(height[0])
```
```
172
```

In [116…
```
# print the first three elements
print(height[0:3])
```
```
[172 159 175]
```

- Arrays can be sorted using the sort function in numpy `np.sort()`:

In [117…
```
np.sort(height)
```
Out[117]:
```
array([159, 167, 172, 173, 175, 180])
```

- Read more about sorting arrays and more advanced array sorting here: https://numpy.org/doc/stable/reference/generated/numpy.sort.html#numpy.sort

Filtering arrays

- height > 170 gives a `Boolean` numpy array.
- `height[height > 170]` gives the values of the array that satisfy `height > 170`.

In [118…
```
print(height)
```
```
[172 159 175 180 167 173]
```

In [119…
```
# returns array with boolean values, where TRUE when height is greater than 170, otherwise FALSE
print(height > 170)
```

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

In [120…

```
# returns array with height values that are greater than 170
print(height[height > 170])
```

```
[172 175 180 173]
```

In [121…

```
# returns array with height values equal to 167
print(height[height == 167])
```

```
[167]
```

In [122…

```
# returns array with height values greater or equal to 172
print(height[height >= 172])
```

```
[172 175 180 173]
```

In [123…

```
# returns array with height values greater than 170 AND height less than 178
print(height[(height > 170) & (height < 178)])
```

```
[172 175 173]
```

In [124…

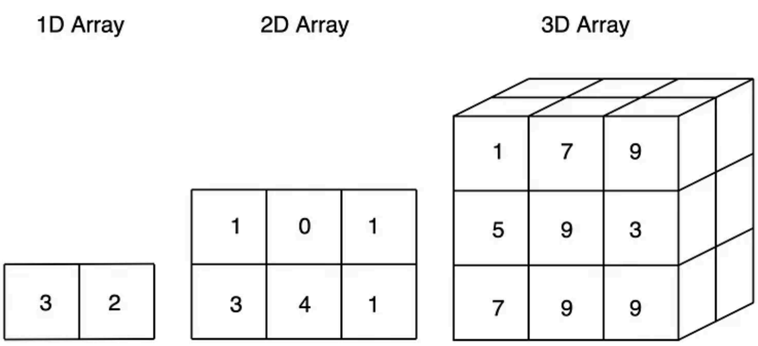
```
# returns array with height values less than 167 OR height more than 178
print(height[(height < 167) | (height > 178)])
```

```
[159 180]
```

Multidimensional numpy arrays

We often talk about an array as if it were a grid in space, with each cell storing one element of the data.

- For instance, if each element of the data were a number, we might visualize a “one-dimensional” array like a list.
- A two-dimensional array would be like a table.
- A three-dimensional array would be like a set of tables, perhaps stacked as though they were printed on separate pages.



In [125…

```
# 0-D array ie scalar
a = np.array(42)
print(a)
```

```
42
```

In [126…

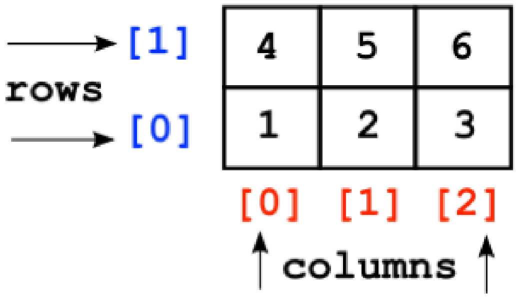
```
# 1-D array with 0-D arrays as its elements
b = np.array([1, 2, 3, 4, 5])
print(b)
```

```
[1 2 3 4 5]
```

In [127…

```
# 2-D array with 1-D arrays as its elements. Often used to represent matrices
c = np.array([[1, 2, 3], [4, 5, 6]])
print(c)
```

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



In [128…

```
# 3-D array with 2-D arrays as its elements
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(d)
```

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

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

Indexing 2D numpy arrays

Use `c.ndim` to find the number of dimensions of the numpy array `c`

In [129…

```
c = np.array([[1, 2, 3], [4, 5, 6]]) # 2-D array
print(c)
```

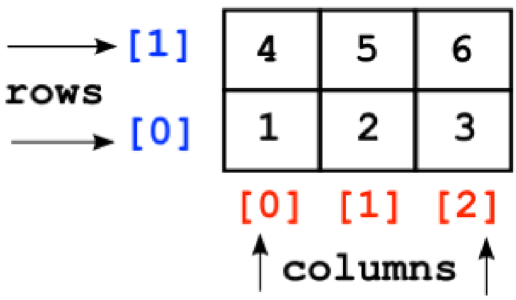
```
[[1 2 3]
 [4 5 6]]
```

In [130…

```
print(c.ndim)
```

```
2
```

Indexing



In [131…

```
# print the second array of the 2-D array
print(c[1]) # gives row 2.
```

```
[4 5 6]
```

In [132…

```
print(c[1][2]) # gives row 1 column 3.
```

```
6
```

In [133…

```
print(c[1,2]) # a more common way to find row 1 column 3.
```

```
6
```

- `<array_name>[row,column]` or `<array_name>[row][column]`
- when using colon symbol `:`, you indicate that you want either all rows or columns or both

- when you specify numbers before (start) and after (stop) colon symbol `:`, it indicates a range (what rows/columns you want).

In [134…
```
print(c[:,0]) # gives all rows and the first and second columns.
```
```
[1 4]
```

In [135…
```
print(c[:,0:2]) # gives all rows and the first and second columns.
```
```
[[1 2]
 [4 5]]
```

In [136…
```
# gives row 1 and all columns in row 1 (same as c[1]).
# it is also the same if c[1,0:] or c[1,]
print(c[1,:])
```
```
[4 5 6]
```

In [137…
```
print(c[-1]) # gives the last row of c.
```
```
[4 5 6]
```

Modifying numpy arrays

In [138…
```
c = np.array([[1, 2, 3], [4, 5, 6]]) # 2-D array
print(c)
```
```
[[1 2 3]
 [4 5 6]]
```

In [139…
```
c[0, 0] = 8
print(c)
```
```
[[8 2 3]
 [4 5 6]]
```

- Be careful with the types in numpy arrays! Example:

In [140…
```
c[0, 0] = 3.6532
print(c)
```
```
[[3 2 3]
 [4 5 6]]
```

Why did this happen?

In [141…
```
print(c.dtype)
```
```
int32
```

- Use `astype` to convert an array from integer to float

In [142…
```
c_float = c.astype(float)
```

In [143…
```
c_float[0, 0] = 3.6532
print(c_float)
```
```
[[3.6532 2.     3.    ]
 [4.     5.     6.    ]]
```

Array operations

Use mathematical operations to compute additions, multiplications, divisions on arrays.

In [144…
```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
```

In [145…
```
a + b # addition on two 1-D arrays
```

Out[145]:
```
array([5, 7, 9])
```

In [146…
```
a*b
```

Out[146]:
```
array([ 4, 10, 18])
```

In [147…
```
print(b/a) # not the same as a/b
```

```
[4.  2.5 2. ]
```

In [148…
```
a*4 # multiply one array by 4
```

Out[148]:
```
array([ 4,  8, 12])
```

In [149…
```
print(a.sum()) # use function sum to sum up all values in array
```

```
6
```

In [150…
```
min(a) # use function min to find minimum value within array
```

Out[150]:
```
1
```

In [151…
```
max(b) # use function max to find maximum value within array
```

Out[151]:
```
6
```

Short Exercises on NumPy (part 1)

1. Import the numpy package as np.
2. Create a numpy array with the first 10 values of the Fibonacci sequence. Then reverse the order of this array (see https://en.wikipedia.org/wiki/Fibonacci_sequence)
3. Find the sum of the fibonacci sequence array you created in exercise 2.
4. Create a 2D array that is a 3x3 matrix.
5. Pick out the first value in the second row. Change its value to be 2 more than the old value.
6. Print the first columns of the 3x3 matrix.
7. Find the sum of each column in the 3x3 matrix.
8. Multiply row 2 in the array by 4.

Statistical functions for numpy arrays

- Can use functions such as `np.mean(c)` to find the mean of all values in the 2-D array c.
- `np.corrcoef()` finds the correlations between the variables.
- `np.std()` calculates the standard deviation.
- `np.column_stack()` is useful for data manipulation. It joins 1-D numpy arrays as columns to make a single 2-D numpy array.

In [152…
```
c = np.array([1, 2, 3, 4, 5, 6])
d = np.array([4, 2, 6, 1, 8, 4])
```

In [153…
```
# column_stack makes a 2D array with 2 columns and 6 rows
#   instead of print(np.array([[1,4], [2,2], [3,6], [4,1], [5,8], [6,4]]))
c_d = np.column_stack((c, d))
print(c_d)
```

```
[[1 4]
 [2 2]
 [3 6]
 [4 1]
 [5 8]
 [6 4]]
```

In [154…
```
print(np.mean(c_d)) # mean for all values within arrays
```

```
3.8333333333333335
```

In [155…
```
np.mean(c_d, axis = 0) # mean by column, where axis = 0 is column and axis = 1 is row
```
Out[155]:
```
array([3.5       , 4.16666667])
```

In [156…
```
np.mean(c_d, axis = 1) # mean by rows
```
Out[156]:
```
array([2.5, 2. , 4.5, 2.5, 6.5, 5. ])
```

In [157…
```
np.sum(c_d, axis = 0) # sum by column
```
Out[157]:
```
array([21, 25])
```

In [158…
```
# correlation coefficients
# It returns a correlation matrix, which shows the correlation coefficients between variables
# for more information https://numpy.org/doc/stable/reference/generated/numpy.corrcoef.html
np.corrcoef(c,d)
```
Out[158]:
```
array([[1.      , 0.271167],
       [0.271167, 1.      ]])
```

- Use `np.unique` to create a new and sorted array with unique values

In [159…
```
c = np.array([1, 2, 3, 1, 5, 6])
d = np.array([4, 2, 6, 4, 8, 4])

c_d = np.column_stack((c, d))
```

In [160…
```
np.unique(c_d)
```
Out[160]:
```
array([1, 2, 3, 4, 5, 6, 8])
```

- If the axis argument isn't passed, your 2D array will be flattened (in other words, from 2-D to 1-D array).
- If you want to get the unique rows or columns, make sure to pass the axis argument. To find the unique rows, specify axis=0 and for columns, specify axis=1.

In [161…
```
# help(np.unique)
```

In [162…
```
print(c_d)
```
```
[[1 4]
 [2 2]
 [3 6]
 [1 4]
 [5 8]
 [6 4]]
```

In [163…
```
np.unique(c_d, axis = 0) # unique rows
```
Out[163]:
```
array([[1, 4],
       [2, 2],
       [3, 6],
       [5, 8],
       [6, 4]])
```

In [164…
```
np.unique(c_d, axis = 1) # unique columns
```
Out[164]:
```
array([[1, 4],
       [2, 2],
       [3, 6],
       [1, 4],
       [5, 8],
       [6, 4]])
```

- To get the indices of unique values in a NumPy array, pass the return_index argument in `np.unique()` as well as your array, using parameter `return_index=True` .

In [165…
```
np.unique(c_d, return_index=True)
# returns a tuple with two arrays,
#  first array is the unique values, and
#  second array is the index of these unique values in original array
```

Out[165]:
```
(array([1, 2, 3, 4, 5, 6, 8]), array([0, 2, 4, 1, 8, 5, 9], dtype=int64))
```

In [166…
```
# help(np.unique)
```

Python Tip

- To "unpacking" a tuple to assign values into different variables, use e.g, `x, y = (1,2)`

In [167…
```
myTuple = (2,3)
x, y = myTuple
print("from myTuple", myTuple, "variable x has value:", x, "and variable y has value:",y)
```

```
from myTuple (2, 3) variable x has value: 2 and variable y has value: 3
```

- Now lets try with `np.unique(array, return_index=True)`

In [168…
```
unique_values, indices_list = np.unique(c_d, return_index=True)
print("unique values in array:", unique_values)
print("index:", indices_list)
```

```
unique values in array: [1 2 3 4 5 6 8]
index: [0 2 4 1 8 5 9]
```

Finding the characteristics of a 3D array

In [169…
```
d = np.array([[[1, 2, 3, 4], [4, 5, 6, 7]],
              [[1, 2, 3, 6], [4, 5, 6, 1]],
              [[0, 2, 4, 6], [1, 3, 5, 7]],
             [[0, 2, 4, 6], [1, 3, 5, 7]]])
print(d)
```

```
[[[1 2 3 4]
  [4 5 6 7]]

 [[1 2 3 6]
  [4 5 6 1]]

 [[0 2 4 6]
  [1 3 5 7]]

 [[0 2 4 6]
  [1 3 5 7]]]
```

In [170…
```
print(d.ndim) # find what dimension the array has
```

```
3
```

In [171…
```
print(d.shape) # finding number of columns, rows, and deep
```

```
(4, 2, 4)
```

- `(4, 2, 4)` : `4` is the number of 2D arrays, `2` is the number of 1D arrays in each of the 2D arrays, and `4` is the length of each 1D array.

In [172…
```
print(d.size) # number of values inside array
```

```
32
```

More advanced indexing of arrays

`d[start:stop:step]` notation

If these are unspecified, they default to the values `start=0` , `stop=end` of array, `step=1`

In [173…
```
d = np.array([4, 2, 6, 1, 8, 4])
```

In [174…
```
print(d[:3]) # get the first 3 elements of d.
```

```
[4 2 6]
```

In [175…
```
print(d[3:]) # get the elements from index 3 onwards
```

```
[1 8 4]
```

In [176…
```
print(d[::3]) # gives every third element beginning at element 0
```

```
[4 1]
```

- Explain the output from `print(d[1::3])` .

In [177…
```
d = np.array([4, 2, 6, 1, 8, 4])
print(d)
```

```
[4 2 6 1 8 4]
```

In [178…
```
print(d[1::3])
```

```
[2 8]
```

Negative step values in array indexing

Negative step values give us an easy way to reverse arrays:

In [179…
```
d = np.array([4, 2, 6, 1, 8, 4])
```

In [180…
```
print(d[::-1])
```

```
[4 8 1 6 2 4]
```

- What do you think `d[4::-3]` gives? Why?

In [181…
```
print(d[4::-3])
```

```
[8 2]
```

The defaults for start and stop swap when negative indexing is use.

In [182…
```
#
# [8, 1, 6, 2, 4, 4]
```

Slicing multi-dimensional arrays

- remember:
  - `array[row, column]` and
  - `array[start_row:end_row, start_column:end_column]`

In [183…
```
people = np.array([[172, 159, 175, 180, 167, 173],
                   [85, 70, 73, 79, 75, 85],
                   [24, 54, 30, 46, 17, 22]])

print(people)
```

```
[[172 159 175 180 167 173]
 [ 85  70  73  79  75  85]
 [ 24  54  30  46  17  22]]
```

In [184…
```
print(people[1, 4])
```

```
75
```

In [185…
```python
print(people[2,:])
```
```
[24 54 30 46 17 22]
```

In [186…
```python
print(people[:2, :4])
```
```
[[172 159 175 180]
 [ 85  70  73  79]]
```

In [187…
```python
print(people[:, ::2])
```
```
[[172 175 167]
 [ 85  73  75]
 [ 24  30  17]]
```

In [188…
```python
print(people[::-1, :])
```
```
[[ 24  54  30  46  17  22]
 [ 85  70  73  79  75  85]
 [172 159 175 180 167 173]]
```

In [189…
```python
print(people[::-1, ::-1])
```
```
[[ 22  17  46  30  54  24]
 [ 85  75  79  73  70  85]
 [173 167 180 175 159 172]]
```

Concatenating arrays using function `concatenate()`

In [190…
```python
c = np.array([1, 2, 3, 4, 5, 6])
d = np.array([4, 2, 6, 1, 8, 4])
e = np.array([7, 10])
```

In [191…
```python
print(np.concatenate([c[0:3], d[0:4], e])) # concatenate arrays
```
```
[ 1  2  3  4  2  6  1  7 10]
```

Stacking arrays

The `concatenate()` function also works on higher dimensional arrays but it can be more clear to use `np.hstack` and `np.vstack`.

In [192…
```python
x = np.array([1, 2, 3])
y = np.array([[9, 8, 7], [6, 5, 4]])
```

In [193…
```python
print(np.vstack([x, y])) # stacks arrays vertically
```
```
[[1 2 3]
 [9 8 7]
 [6 5 4]]
```

In [194…
```python
y = np.array([[9, 8, 7], [6, 5, 4]])
z = np.array([[6], [3]])
```

In [195…
```python
print(np.hstack([y, z])) # stacks arrays horizontally
```
```
[[9 8 7 6]
 [6 5 4 3]]
```

In [196…
```python
y_list = y.tolist()

print(y_list)
print(type(y_list))
```
```
[[9, 8, 7], [6, 5, 4]]
<class 'list'>
```

- Where could we run into errors with `vstack` and `hstack`?

Views and copies of arrays

- Array slices return views rather than copies of the array data. This is one area in which Numpy array slicing differs from Python list slicing: in lists, slices will be copies.
- If array slices are stored as a variable and modified, the original array will also be modified.
- This can be useful to work on small subsets of large arrays but can be **dangerous!**

Example:

In [197…
```
d = np.array([4, 2, 6, 1, 8, 4])
print(d)
```

```
[4 2 6 1 8 4]
```

In [198…
```
d_slice = d[2:]
d_slice[0] = 10
print(d)
```

```
[ 4  2 10  1  8  4]
```

Use the `.copy()` function when slicing an array to make a copy of the slice, in effect creating a new array that does not affect the original array when it is modified.

Example:

In [199…
```
d_slice = d[2:].copy() # make a copy of the slice array d
d_slice[0] = 10
print(d)
print(d_slice)
```

```
[ 4  2 10  1  8  4]
[10  1  8  4]
```

Using `arr.reshape()` will give a new shape to an array without changing the data. Just remember that when you use the reshape method, the array you want to produce needs to have *the same number* of elements as the original array. If you start with an array with 12 elements, you'll need to make sure that your new array also has a total of 12 elements.

In [200…
```
a = np.arange(6) # create an array of 6 elements from 0 to 5
print(a)
```

```
[0 1 2 3 4 5]
```

In [201…
```
print(a.reshape(3, 2))
```

```
[[0 1]
 [2 3]
 [4 5]]
```

In [202…
```
print(a.reshape(2,3))
```

```
[[0 1 2]
 [3 4 5]]
```

In [203…
```
# print(a.reshape(3,3)) # error, why?
```

Converting numpy arrays to lists

- Notice that the elements of numpy arrays all have the same data type.
- To convert a numpy array to a list use the `.tolist()` command.
- You can also use the function `list()` to convert numpy array to list.
- Use `random.randint()` to create an array with random integers

Example:

In [204…
```
# lets create an array with five random integers from 1 to 5
rolls = np.random.randint(low = 1, high = 6, size = 5)
print(rolls)
print(type(rolls))
```

```
[3 2 1 4 4]
<class 'numpy.ndarray'>
```

In [205…

```
rolls_list = rolls.tolist()
print(type(rolls_list))
```

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

In [206…

```
rolls_list2 = list(rolls)
print(type(rolls_list2))
```

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

Reading files in as Numpy arrays

- There are methods for reading files containing numeric data in to Python as Numpy arrays.
- `np.loadtxt()` and `np.genfromtxt()` can be used:
- However, `Pandas` makes the reading in process much easier, and is better for mixed data types.
    - Pandas `DataFrames` are a more usable format.
- We will focus more on reading data in using Pandas.

In [207…

```
filename = 'C:/Users/cepedazk/Jupyter Notebook/Datasets/Data.txt'
```

In [208…

```
mydata = np.loadtxt(filename, delimiter = ',', skiprows = 1, dtype = str)
print(mydata)
```

```
['"1" "1" "1" "1.78" "77.7" "85" "N" "119" "1" "75.2"'
 '"2" "3" "1" "1.68" "74.9" "103" "N" "131" "4" "75.9"'
 '"3" "3" "1" "1.89" "82.8" "103" "D" "122" "4" "82.7"'
 '"4" "1" "1" "2.02" "90.2" "103" "N" "144" "2" "89.8"'
 '"5" "3" "1" "1.9" "92" "91" "N" "132" "4" "91.9"'
 '"6" "1" "1" "1.92" "81.2" "108" "N" "140" "1" "78.9"'
 '"7" "3" "1" "1.71" "72.3" "101" "N" "109" "4" "76.7"'
 '"8" "1" "1" "1.9" "79.4" "121" "N" "157" "3" "79.5"'
 '"9" "3" "1" "1.76" "81.4" "109" "N" "122" "4" "83.9"'
 '"10" "1" "1" "1.99" "83.8" "111" "N" "130" "2" "84.3"'
 '"11" "3" "1" "1.78" "82.2" "79" "D" "148" "4" "81.8"'
 '"12" "3" "1" "1.59" "69.1" "96" "N" "101" "4" "70.5"'
 '"13" "1" "1" "1.71" "75.7" "118" "D" "133" "1" "77.1"'
 '"14" "2" "1" "1.82" "85.2" "90" "N" "153" "3" "83.3"'
 '"15" "2" "1" "1.57" "70" "81" "N" "115" "3" "70.4"'
 '"16" "3" "1" "1.8" "82" "107" "N" "138" "5" "85.1"'
 '"17" "1" "1" "1.67" "79.6" "105" "N" "141" "1" "81.3"'
 '"18" "3" "1" "1.68" "65.2" "114" "N" "109" "4" "67.2"'
 '"19" "1" "1" "1.66" "68.5" "105" "N" "112" "1" "63.6"'
 '"20" "2" "1" "1.94" "84.6" "78" "D" "139" "3" "85.3"'
 '"21" "3" "1" "1.88" "83.8" "108" "N" "150" "4" "84.6"'
 '"22" "3" "1" "1.67" "76.9" "67" "D" "103" "4" "78.8"'
 '"23" "2" "1" "1.87" "91.3" "104" "N" "177" "3" "91.8"'
 '"24" "3" "1" "1.58" "72.3" "108" "N" "122" "4" "73.8"'
 '"25" "3" "1" "1.67" "76.5" "82" "N" "111" "4" "75.5"'
 '"26" "1" "1" "1.75" "69.8" "100" "N" "101" "2" "71"'
 '"27" "1" "1" "1.67" "72.2" "86" "D" "127" "3" "71.7"'
 '"28" "2" "1" "1.94" "84.2" "94" "D" "127" "3" "83.5"'
 '"29" "1" "1" "1.65" "75.7" "101" "D" "133" "1" "74"'
 '"30" "3" "1" "1.77" "78.3" "98" "D" "166" "4" "81.3"'
 '"31" "3" "2" "1.57" "72.8" "82" "N" "112" "4" "75.3"'
 '"32" "3" "2" "1.66" "75.1" "119" "N" "145" "4" "71.3"'
 '"33" "1" "2" "1.48" "65.3" "94" "N" "107" "2" "63.7"'
 '"34" "1" "2" "1.46" "61.7" "93" "N" "79" "1" "61.5"'
 '"35" "3" "2" "1.62" "67" "91" "N" "92" "4" "69.8"'
 '"36" "2" "2" "1.68" "70.1" "135" "N" "116" "3" "70"'
 '"37" "3" "2" "1.85" "83.8" "85" "N" "152" "4" "81.9"'
 '"38" "1" "2" "1.45" "65.7" "68" "N" "102" "1" "64.1"'
 '"39" "3" "2" "1.78" "78.5" "95" "N" "144" "5" "73.7"'
 '"40" "1" "2" "1.68" "73.6" "108" "N" "146" "2" "71.3"'
 '"41" "1" "2" "1.81" "78.3" "96" "N" "126" "1" "76.9"'
 '"42" "3" "2" "1.85" "82.2" "84" "N" "145" "4" "82.6"'
 '"43" "3" "2" "1.79" "68.3" "72" "N" "129" "4" "66.1"'
 '"44" "3" "2" "1.5" "63.8" "122" "N" "61" "4" "63.2"'
 '"45" "3" "2" "1.47" "59" "77" "N" "107" "4" "60.4"'
 '"46" "1" "2" "1.41" "58.3" "92" "N" "96" "2" "57.1"'
 '"47" "3" "2" "1.66" "72.9" "82" "N" "135" "5" "73.7"'
 '"48" "3" "2" "1.65" "74.9" "118" "N" "129" "4" "73.5"'
 '"49" "3" "2" "1.73" "64.4" "117" "N" "108" "4" "62.4"'
 '"50" "2" "2" "1.62" "67.3" "99" "D" "72" "3" "66.5"'
 '"51" "1" "2" "1.84" "77.7" "104" "N" "141" "2" "78.4"'
 '"52" "2" "2" "1.61" "73.7" "100" "N" "122" "3" "71.4"'
 '"53" "2" "2" "1.35" "64" "94" "N" "84" "3" "64.8"'
 '"54" "3" "2" "1.76" "77.3" "110" "N" "123" "4" "78"'
 '"55" "3" "2" "1.83" "79.1" "107" "D" "138" "4" "79.8"'
 '"56" "2" "2" "1.52" "63.3" "125" "D" "119" "3" "65.5"'
 '"57" "1" "2" "1.57" "68.7" "105" "N" "113" "2" "68.8"'
 '"58" "3" "2" "1.69" "68.1" "122" "D" "95" "5" "66.6"'
 '"59" "1" "2" "1.8" "77.1" "94" "N" "124" "1" "76.3"'
 '"60" "3" "2" "1.79" "75.3" "95" "N" "144" "5" "74.4"']
```

Numpy exercise (Part 2)

1. Import the numpy package as np.
2. Create a numpy array with the first 10 values of the Fibonacci sequence. Then reverse the order of this array.
3. Create another array with 10 random integers between 1 and 40.
4. Find the sum of the random array.
5. Add the array of 10 random integers to the reversed array of the Fibonacci sequence element wise.
6. Create a 2D array that is a 3x3 matrix.
7. Pick out the first value in the second row. Change its value to be 2 more than the old value.
8. Pick out the first and third columns of the 3x3 matrix.

9. Use np.hstack to add another column to the array.
10. Find the sum of each column in the array.
11. Multiply row 2 in the array by row 4 element wise.
12. Reverse the order of the rows in the array
13. Create a copy of your array.
14. Convert your Numpy array to a list.