Data Analysis with Python

Intro to NumPy

Aim: learn why **NumPy** is an important library for the data-processing world in Python

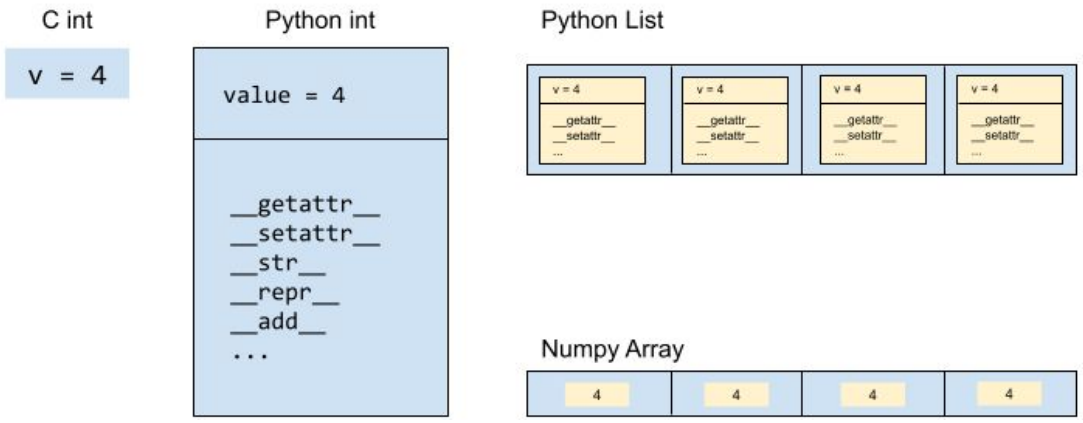
- NumPy provides the following:
    - Computations, memory storage, illustrates how Excel will always be limited when processing large volumes of data and more

Numpy: Numeric computing library

NumPy (Numerical Python): one of the core packages for numerical computing in Python, Pandas, Matplotlib, Statmodels

- many other scientific libraries rely on NumPy
- Major contributions:
    - Efficient numerical computation with C primitives
    - Efficient collections with vectorized operations
    - An integrated and natural Linear Algebra API
    - A C API for connecting NumPy with libraries written in C, C++, or FORTRAN

Let's develop on efficiency. In Python, **everything is an object**, which means that even simple ints are also objects, with all the required machinery to make object work. We call them "Boxed Ints". In contrast, NumPy uses primitive numeric types (floats, ints) which makes storing and computation efficient.



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

import sys
import numpy as np

Basic Numpy Arrays

```
In [3]:
# Basic Numpy Arrays
np.array([1, 2, 3, 4])

array([1, 2, 3, 4])
```

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

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

In [5]:
b = np.array([0, .5, 1, 1.5, 2])
```

a = np.array([1, 2, 3, 4])
b = np.array([0, .5, 1, 1.5, 2])

```
In [6]:
a[0], a[1]

(1, 2)
```

a[0], a[1]

```
In [7]:
a[0:]

array([1, 2, 3, 4])
```

a[0:]

- Provides all entries from 0 index up to AND including the max index.
- Note: compared to the output above (*In [6]:*), *In[7]:* has a **range**
    - Specifically '**:**' results in output containing array([...])
    - While *In[6]:*, extracting entries within certain indices provides (1, 2)
        - generally , (...)

```
In [8]: a[1:3]
        # Indices from 1 up to 3, excluding index 3.

        array([2, 3])
```

a[1:3]

```
In [9]: a[1:-1]

        array([2, 3])
```

a[1:-1]

```
In [10]: a[::2]

         array([1, 3])
```

a[::2]

- The index entries right before AND after *2*

a[::int]

- Generally, the entries right before and after the specified int

```
In [11]: b

         array([0. , 0.5, 1. , 1.5, 2. ])

In [12]: b[0], b[2], b[-1]

         (0.0, 1.0, 2.0)
```

b[0], b[2], b[-1]

```
In [14]: b[[0, 2, -1]]

         array([0., 1., 2.])
```

b[[0, 2, -1]]

- Compared to the code above (*In[12]*), *In[14]* provides array([...])
- While *In[12]* provides (...)
- Generally, b[index]
  - Provides (...)

- And b[[...]]
    - Provides array([...])

Array Types

```
In [16]: a
Out[16]: array([1, 2, 3, 4])
In [17]: a.dtype
Out[17]: dtype('int32')
```

a.dtype

```
In [18]: b
Out[18]: array([0. , 0.5, 1. , 1.5, 2. ])
In [19]: b.dtype
Out[19]: dtype('float64')
```

b.dtype

```
In [21]: np.array([1, 2, 3, 4], dtype=float)
Out[21]: array([1., 2., 3., 4.])
```

np.array([1, 2, 3, 4], dtype=float)

```
In [23]: np.array([1, 2, 3, 4], dtype=np.int8)
Out[23]: array([1, 2, 3, 4], dtype=int8)
```

```
In [24]: c = np.array(['a', 'b', 'c'])
```

c = np.array([‘a’, ‘b’, ‘c’])

```
In [25]: c.dtype

dtype('<U1')
```

c.dtype

```
In [26]: d = np.array([{'a': 1}, sys])

In [27]: d.dtype

dtype('O')
```

d = np.array([{'a': 1}, sys])
d.dtype

Dimensions and shapes

```
In [28]: # Dimensions and shapes
         A = np.array([
             [1, 2, 3],
             [4, 5, 6]
         ])
```

A = np.array([
[1, 2, 3],
[4, 5, 6]
])

```
In [30]: A.shape
         # (2 rows, 3 columns)

(2, 3)
```

A.shape

```
In [40]: A.ndim
         # 2 sets of [[?

2
```

A..ndim

```
In [34]:
B = np.array([
    [
        [12, 11, 10],
        [9, 8, 7]
    ],
    [
        [6, 5, 4],
        [3, 2, 1]
    ]
])
```

B = np.array([
[
[12, 11, 10],
[9, 8, 7]
],
[
[6, 5, 4],
[3, 2, 1]
]
])

```
In [35]:
B

array([[[12, 11, 10],
        [ 9,  8,  7]],

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

```
In [37]:
B.shape
# 2 groups, 2 rows in each group, 3 columns in each group

(2, 2, 3)
```

B.shape

```
In [39]: B.ndim

3
```

B.ndim

```
In [42]: B.size
         # 12 individual numbers within the array.

12
```

B.size

```
In [44]: C = np.array([
             [
                 [12, 11, 10],
                 [9, 8, 7],
             ],
             [
                 [6, 5, 4]
             ]
         ])
```

```
In [45]: C.dtype

dtype('O')
```

C.dtype

```
In [46]: C.shape

(2,)
```

C.shape

- Take a look at what *C* looks like when prompted

```
In [167]: C

array([list([[12, 11, 10], [9, 8, 7]]), list([[6, 5, 4]])], dtype=object)
```

```
In [47]: C.size

2
```

C.size

- *2* b/c C consists of 2 lists

```
In [48]: type(C[0])

list
```

type(C[0])

- The first object within *C* is of type *list*

Indexing and Slicing of Matrices

Note: Indices START @ 0.

```
# Square matrix
A = np.array([
#.   0. 1. 2.
    [1, 2, 3], #0
    [4, 5, 6], #1
    [7, 8 ,9]  #2
])
```

A = np.array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
])

```
In [59]: A[1]

array([4, 5, 6])
```

A[1]

```
In [60]: A[1][0]
         # row first, then column.

Out[60]: 4
```

A[1][0]

```
In [61]: # A[d1, d2, d3, d4]
         A[1, 0]
         # basically combined the above.

Out[61]: 4
```

A[1, 0]

```
In [62]: A[0:2]
         # All rows from index 0(including 0) up to and excluding index 2.

Out[62]: array([[1, 2, 3],
                [4, 5, 6]])
```

A[0:2]

- Notice the difference between *In[59]* above and *In[62]*
  - *In[62]* provides a range
    - And array([[1, 2, 3], …]]) as output
  - While *In[59]* provides a basic slice/ index
    - And array([4, 5, 6]) as output

```
In [63]: A[:, :2]

Out[63]: array([[1, 2],
                [4, 5],
                [7, 8]])
```

A[:, :2]

```
In [64]: A[:2, 2:]

Out[64]: array([[3],
                [6]])
```

A[:2, 2:]

```
In [65]: A

array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [66]: A[2] = 99
         # This updates all indices in 2 position as 99.

In [67]: A

array([[ 1,  2,  3],
       [ 4,  5,  6],
       [99, 99, 99]])
```

A[2] = 99

Summary Statistics

```
In [68]: # Summary Statistics
         a = np.array([1, 2, 3, 4])
```

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

```
In [69]: a.sum()

10
```

a.sum()

```
In [70]: a.mean()

2.5
```

a.mean()

```
In [71]: a.std()

1.118033988749895
```

a.std()

```
In [72]: a.var()

1.25
```

a.var()

```
In [73]: A = np.array([
             [1, 2, 3],
             [4, 5, 6],
             [7, 8, 9]
         ])
```

A = np.array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
])

```
In [74]: A.sum()

45

In [75]: A.mean()

5.0

In [76]: A.std()

2.581988897471611

In [77]: A.var()

6.666666666666667
```

```
In [78]:
A.sum(axis=0)
# This provides the sum of numbers within each column index.

array([12, 15, 18])
```

A.sum(axis=0)

- Note the difference between In[74] - In[77] and In[78]
  - *In[74] - In[77]* provides output that takes into consideration the array as a whole
  - *In[78]* provides array([12, 15, 18]) as output

```
In [79]:
A.sum(axis=1)
# This provides the sum of numbers within each row index.

array([ 6, 15, 24])
```

A.sum(axis=1)

Generally, Matrix.sum(axis=0)

- Provides the sum of numbers within each **column** index
  array([sumOfcol1, sumOfcol2, sumOfcol3])
- While Matrix.sum(axis=1) provides the sum of numbers within each **row**
  array([sumOfrow1, sumOfrow2, sumOfrow3])

```
In [80]:
A.mean(axis=0)
# This provides the mean/ avrg. of the numbers within each column index.
# Ex. For column 1: 1+4+7 = 12 / 3 = 4 <- which is the avrg.

array([4., 5., 6.])

In [ ]:
A.mean(axis=1)
# Similar to above.
# This provides the mean/avg. of the numbers within each row index.
# Ex. For row 1: 1+2+3 = 6 / 3 = 2
```

A.mean(axis=0)

A.mean(axis=1)

```
In [81]:
A.std(axis=0)

array([2.44948974, 2.44948974, 2.44948974])

In [82]:
A.std(axis=1)

array([0.81649658, 0.81649658, 0.81649658])
```

A.std(axis=0)

A.std(axis=1)

Generally,

- Matrix.function(axis=0)
  - Provides the output of the function for each **column i**ndex
- Matrix.function(axis=1)
  - Provides the output of the function for each **row** index

Broadcasting and Vectorized Operations

```
In [84]: # Broadcasting and Vectorized Operations

         a = np.arange(4)
         # Creates an array consisting of 4 entries as the indices.
```

a = np.arange(4)

```
In [85]: a

         array([0, 1, 2, 3])

In [86]: a + 10

         array([10, 11, 12, 13])

In [87]: a * 10

         array([ 0, 10, 20, 30])
```

a + 10
a * 10

```
In [88]: a

         array([0, 1, 2, 3])
```

- Notice when *a* is prompted, *a* is **NOT** modified
- None of *In[86]* and *In[87]* modified *a*

```
In [89]: a += 100

In [90]: a

Out[90]: array([100, 101, 102, 103])
```

a += 100

- Notice that *In[89]* MODIFIED *a*
- The presence of = MODIFIED *a*

```
In [91]: l = [0, 1, 2, 3]

In [92]: [i * 10 for i in l]

Out[92]: [0, 10, 20, 30]
```

l = [0, 1, 2, 3]
[i * 10 for i in l]

```
In [93]: a = np.arange(4)

In [94]: a

Out[94]: array([0, 1, 2, 3])
```

a = np.arange(4)

```
In [95]: b = np.array([10, 10, 10, 10])

In [96]: b

Out[96]: array([10, 10, 10, 10])
```

b = np.array([10, 10, 10, 10])

```
In [97]: a + b

Out[97]: array([10, 11, 12, 13])

In [98]: a * b

Out[98]: array([ 0, 10, 20, 30])
```

a + b
a * b

Boolean Arrays

```
In [99]:  # Boolean Arrays
          a = np.arange(4)
In [100]: a
          array([0, 1, 2, 3])
```

a = np.arange(4)

```
In [101]: a[0], a[-1]
          (0, 3)
```

a[0], a[-1]

```
In [102]: a[[0, -1]]
          array([0, 3])
```

a[[0, -1]]

- Notice the difference between *In[101]* and *In[102]*
    - In terms of output as well

```
In [103]: a[[True, False, False, True]]
          array([0, 3])
```

a[[True, False, False, True]]

```
In [104]: a
          array([0, 1, 2, 3])
In [105]: a >= 2
          array([False, False,  True,  True])
```

a >= 2

```
In [106]: a[a >= 2]

          array([2, 3])
```

a[a >= 2]

- Notice the difference between *In[105]* and *In[106]*
- *In[105]* only consists of operators (>=)
    - This provides an array of booleans as output
        - General output: array([list of booleans...])
- While *In[106]* consists of operators and list / index / slicing brackets (*[ ]*)
    - This provides an array of the entries within certain indices that satisfy the conditions / operators as output
        - General output: array([entry 1, entry 2, entry3, entryn...])

```
In [107]: a.mean()

          1.5
```

a.mean()

```
In [108]: a[a > a.mean()]

          array([2, 3])
```

a[a > a.mean()]

```
In [109]: a[~(a > a.mean())]
          # The '~' implies negation of the conditions within the brackets.

          array([0, 1])
```

a[~(a > a.mean())]

```
In [110]: a[(a == 0) | (a == 1)]

          array([0, 1])
```

a[(a == 0) | (a == 1)]

```
In [111]: a[(a <= 2) & (a % 2 == 0)]

          array([0, 2])
```

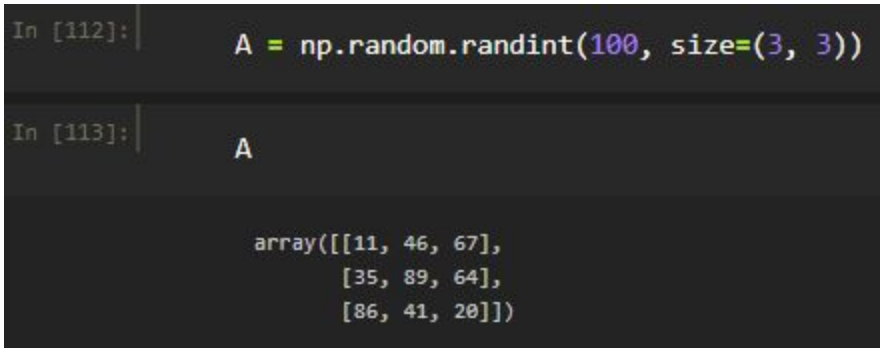
a[(a <= 2) & (a % 2 == 0)]

Generally, if the prompt / code consists of ONLY comparison operators (>, <, >=, etc...)

- Then array([boolean1, boolean2, boolean3, boolean,...]) will be the output

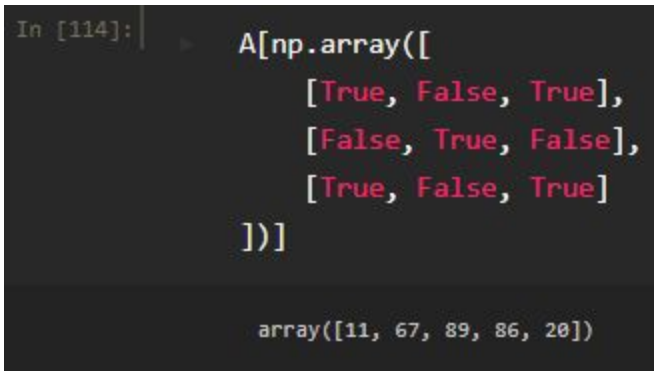
If the prompt / code consists of comparison operators within a list / square brackets (*[...]*)

- Then an array of the entries within certain indices that satisfy the conditions / operators will be provided as output
- General output: array([entry 1, entry 2, entry3, entryn...])



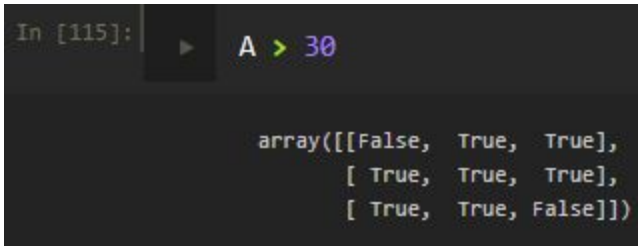
A = np.random.randint(100, size=(3, 3))

- This provides a 3 by 3 array (3 rows, 3 columns) consisting of random numbers within each entry and each number is less than 100



A[np.array([
[True, False, True],
[False, True, False],
[True, False, True]
])]

- This provides an array consisting of entries of indices that satisfy *True*



A > 30

```
In [116]: A[A > 30]

array([46, 67, 35, 89, 64, 86, 41])
```

A[A > 30]

- Notice the difference between *In[115]* and *In[116]*

Generally,

If a prompt / code consists of ONLY a condition operator

- Then it will provide an array consisting of booleans as output
- ALSO **NOTE** that the output will consist of an extra pair of square brackets (*[ ]*)
- Output:

array([[boolean1, boolean2, boolean3, booleann,...]
[.....]
[.....]
]])

If a prompt / code consists of a condition operator within square brackets

- Then it will provide a basic array consisting of entries that satisfy the specified conditions
- Output:

array([entry1, entry2, entry3, entryn, ...])

Linear Algebra

```
In [117]: # Linear Algebra
          A = np.array([
              [1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]
          ])
```

A = np.array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
])

```
In [118]: B = np.array([
              [6, 5],
              [4, 3],
              [2, 1]
          ])
```

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

```
In [119]: A.dot(B)

array([[20, 14],
       [56, 41],
       [92, 68]])
```

A.dot(B)

- Matrix multiplication (Matrix A x Matrix B)

```
In [120]: A @ B

array([[20, 14],
       [56, 41],
       [92, 68]])
```

A @ B

```
In [121]: B.T

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

B.T

- Matrix B multiplied by its Transpose

```
In [122]: A

array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [123]: B.T @ A

array([[36, 48, 60],
       [24, 33, 42]])
```

B.T @ A

- The product of B.T multiplied by matrix A

References

https://www.youtube.com/watch?v=r-uOLxNrNk8

https://github.com/ine-rmotr-curriculum/freecodecamp-intro-to-numpy/blob/master/2.%20NumPy.ipynb

https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.ndarray.html#array-methods