This notebook explores some more and a bit advanced methods available in NumPy.

1. Stacking: Several arrays can be stacked together along different axes.
2. np.vstack: To stack arrays along vertical axis.
3. np.hstack: To stack arrays along horizontal axis.
4. np.column_stack: To stack 1-D arrays as columns into 2-D arrays.
5. np.concatenate: To stack arrays along specified axis (axis is passed as argument).

In [1]:

```
import numpy as np

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

b = np.array([[5, 6],
              [7, 8]])

# vertical stacking
print("Vertical stacking:\n", np.vstack((a, b)))
```

```
Vertical stacking:
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

In [2]:

```
# horizontal stacking
print("\nHorizontal stacking:\n", np.hstack((a, b)))
```

```
Horizontal stacking:
 [[1 2 5 6]
 [3 4 7 8]]
```

In [3]:

```
c = [5, 6]

# stacking columns
print("\nColumn stacking:\n", np.column_stack((a, c)))
```

```
Column stacking:
 [[1 2 5]
 [3 4 6]]
```

1. Splitting:For splitting, we have these functions:

np.hsplit:

Split array along horizontal axis.

np.vsplit:

Split array along vertical axis.

np.array_split:

Split array along specified axis.

In [5]:

```
import numpy as np

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

# horizontal splitting
print("Splitting along horizontal axis into 2 parts:\n", np.hsplit(a, 2))
```

```
Splitting along horizontal axis into 2 parts:
 [array([[1, 3, 5],
       [2, 4, 6]]), array([[ 7,  9, 11],
       [ 8, 10, 12]])]
```

In [6]:

```
# vertical splitting
print("\nSplitting along vertical axis into 2 parts:\n", np.vsplit(a, 2))
```

```
Splitting along vertical axis into 2 parts:
 [array([[ 1,  3,  5,  7,  9, 11]]), array([[ 2,  4,  6,  8, 10, 12]])]
```

3. Broadcasting:

The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes.

In [7]:

```
import numpy as np

a = np.array([1.0, 2.0, 3.0])

# Example 1
b = 2.0
print(a * b)
```

```
[2. 4. 6.]
```

In [8]:

```
# Example 2
c = [2.0, 2.0, 2.0]
print(a * c)
```

```
[2. 4. 6.]
```

We can think of the scalar b being stretched during the arithmetic operation into an array with the same shape as a. The new elements in b, as shown in above figure, are simply copies of the original scalar. Although, the stretching analogy is only conceptual. Numpy is smart enough to use the original scalar value without actually making copies so that broadcasting operations are as memory and computationally efficient as possible. Because Example 1 moves less memory, (b is a scalar, not an array) around during the multiplication, it is about 10% faster than Example 2 using the standard numpy on Windows 2000 with one million element arrays! The figure below makes the concept more clear:

In above example, the scalar b is stretched to become an array of with the same shape as a so the shapes are compatible for element-by-element multiplication.

Now, let us see an example where both arrays get stretched.

In [10]:

```
import numpy as np

a = np.array([0.0, 10.0, 20.0, 30.0])
b = np.array([0.0, 1.0, 2.0])

print(a[:, np.newaxis] + b)
```

```
[[ 0.  1.  2.]
 [10. 11. 12.]
 [20. 21. 22.]
 [30. 31. 32.]]
```

In some cases, broadcasting stretches both arrays to form an output array larger than either of the initial arrays.

4. Working with datetime:

> Numpy has core array data types which natively support datetime functionality. T he data type is called “datetime64”, so named because “datetime” is already take n by the datetime library included in Python.

Consider the example below for some examples:

In [11]:

```
import numpy as np

# creating a date
today = np.datetime64('2017-02-12')
print("Date is:", today)
print("Year is:", np.datetime64(today, 'Y'))
```

```
Date is: 2017-02-12
Year is: 2017
```

In [12]:

```
# creating array of dates in a month
dates = np.arange('2017-02', '2017-03', dtype='datetime64[D]')
print("\nDates of February, 2017:\n", dates)
print("Today is February:", today in dates)
```

```
Dates of February, 2017:
 ['2017-02-01' '2017-02-02' '2017-02-03' '2017-02-04' '2017-02-05'
 '2017-02-06' '2017-02-07' '2017-02-08' '2017-02-09' '2017-02-10'
 '2017-02-11' '2017-02-12' '2017-02-13' '2017-02-14' '2017-02-15'
 '2017-02-16' '2017-02-17' '2017-02-18' '2017-02-19' '2017-02-20'
 '2017-02-21' '2017-02-22' '2017-02-23' '2017-02-24' '2017-02-25'
 '2017-02-26' '2017-02-27' '2017-02-28']
Today is February: True
```

In [13]:

```
# arithmetic operation on dates
dur = np.datetime64('2017-05-22') - np.datetime64('2016-05-22')
print("\nNo. of days:", dur)
print("No. of weeks:", np.timedelta64(dur, 'W'))
```

```
No. of days: 365 days
No. of weeks: 52 weeks
```

In [14]:

```
# sorting dates
a = np.array(['2017-02-12', '2016-10-13', '2019-05-22'], dtype='datetime64')
print("\nDates in sorted order:", np.sort(a))
```

```
Dates in sorted order: ['2016-10-13' '2017-02-12' '2019-05-22']
```

Linear algebra in NumPy:

The Linear Algebra module of NumPy offers various methods to apply linear algebra on any numpy array.

You can find:

1. rank, determinant, trace, etc. of an array.
2. eigen values of matrices
3. matrix and vector products (dot, inner, outer,etc. product), matrix exponentiation
4. solve linear or tensor equations and much more!

Consider the example below which explains how we can use NumPy to do some matrix operations.

In [15]:

```
import numpy as np

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

print("Rank of A:", np.linalg.matrix_rank(A))
```

```
Rank of A: 3
```

In [16]:

```
print("\nTrace of A:", np.trace(A))
```

```
Trace of A: 11
```

In [17]:

```
print("\nDeterminant of A:", np.linalg.det(A))
```

```
Determinant of A: -306.0
```

In [18]:

```
print("\nInverse of A:\n", np.linalg.inv(A))
```

```
Inverse of A:
 [[ 0.17647059 -0.00326797 -0.02287582]
 [ 0.05882353 -0.13071895  0.08496732]
 [-0.11764706  0.1503268   0.05228758]]
```

In [19]:

```
print("\nMatrix A raised to power 3:\n", np.linalg.matrix_power(A, 3))
```

```
Matrix A raised to power 3:
 [[336 162 228]
 [406 162 469]
 [698 702 905]]
```