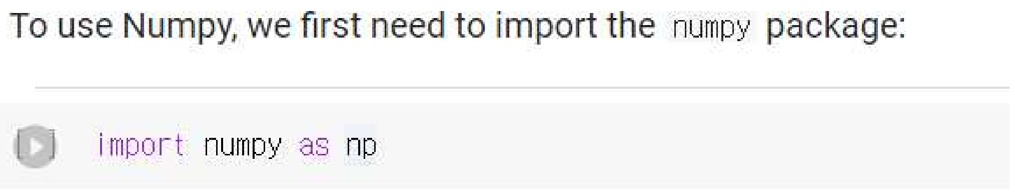
Numpy

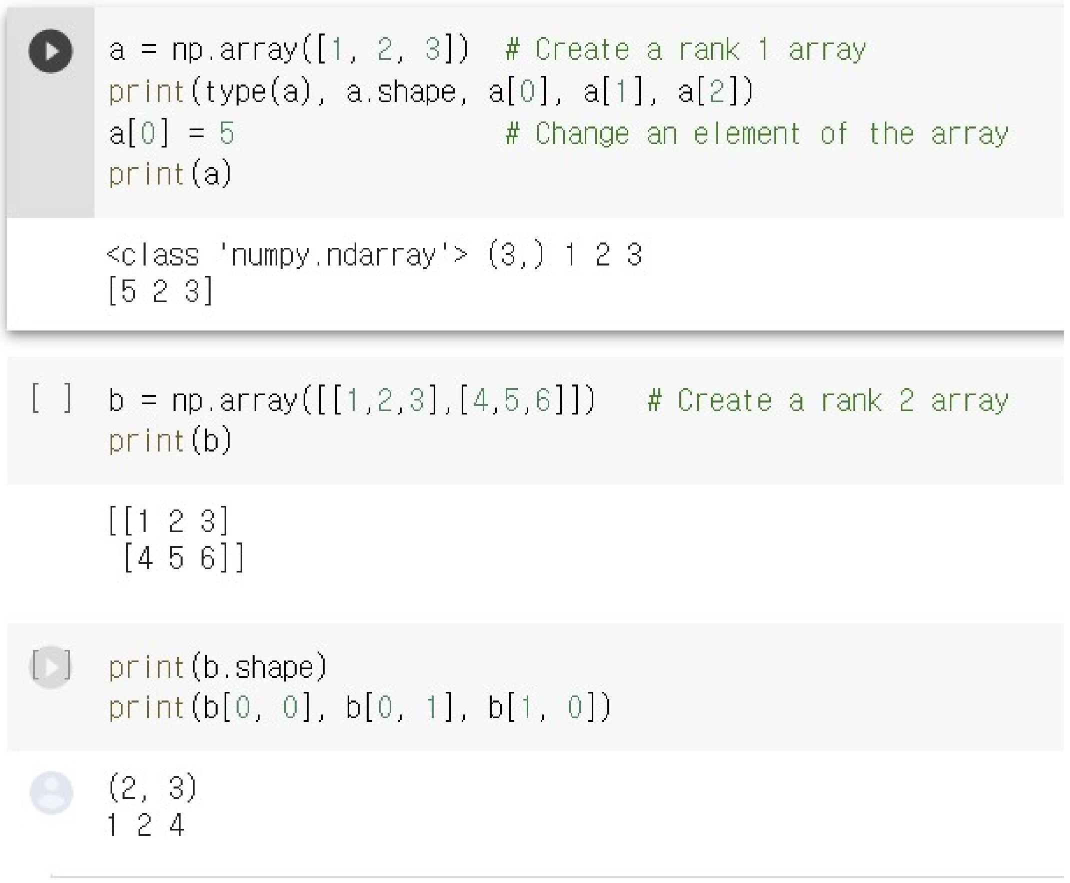
- Numpy is the core library for scientific computing in Python.
- It provides a high-performance multidimensional array object, and tools for working with these arrays.



To use Numpy, we first need to import the `numpy` package:

```
import numpy as np
```

Numpy: Array



```
a = np.array([1, 2, 3])  # Create a rank 1 array
print(type(a), a.shape, a[0], a[1], a[2])
a[0] = 5                 # Change an element of the array
print(a)
```

```
<class 'numpy.ndarray'> (3,) 1 2 3
[5 2 3]
```

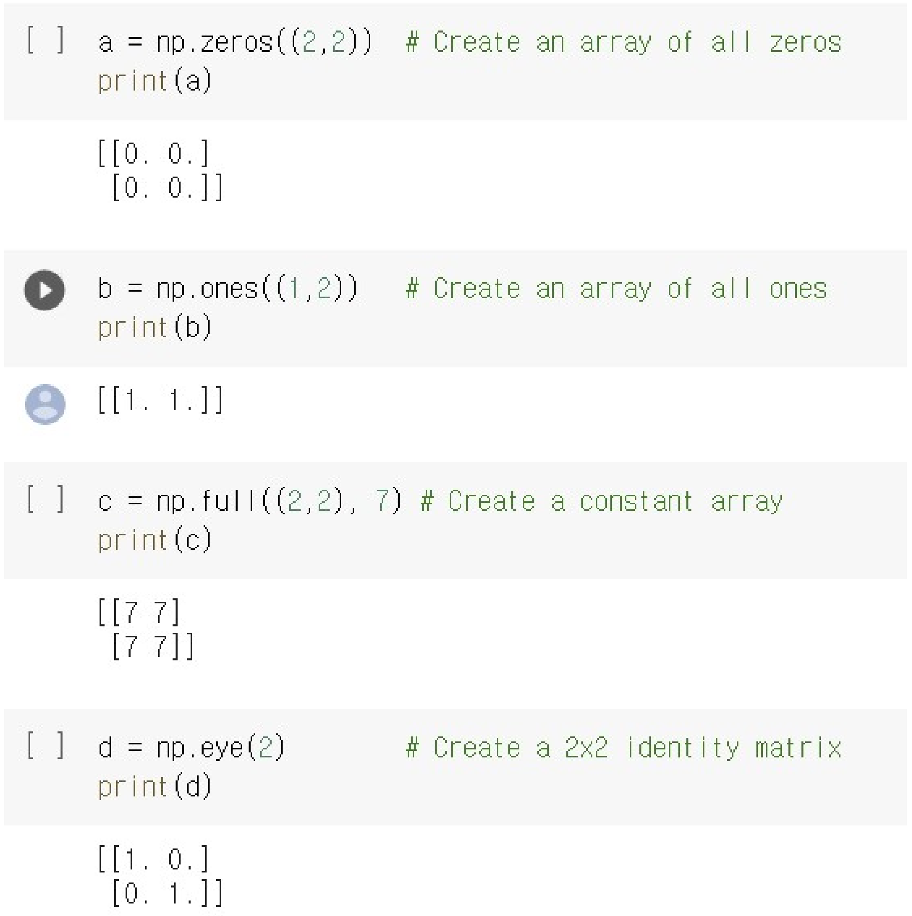
```
b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b)
```

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

```
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

```
(2, 3)
1 2 4
```

Numpy: Array



```
a = np.zeros((2,2))   # Create an array of all zeros
print(a)
```

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

```
b = np.ones((1,2))    # Create an array of all ones
print(b)
```

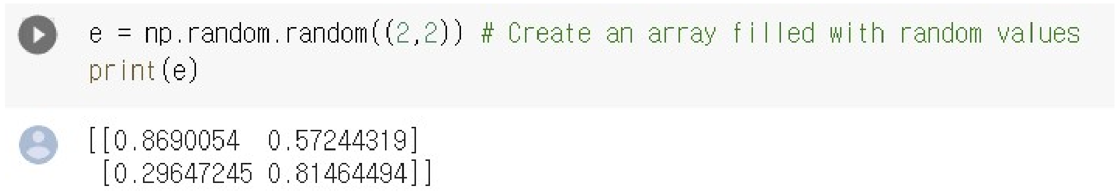
```
[[1. 1.]]
```

```
c = np.full((2,2), 7) # Create a constant array
print(c)
```

```
[[7 7]
 [7 7]]
```

```
d = np.eye(2)         # Create a 2x2 identity matrix
print(d)
```

```
[[1. 0.]
 [0. 1.]]
```

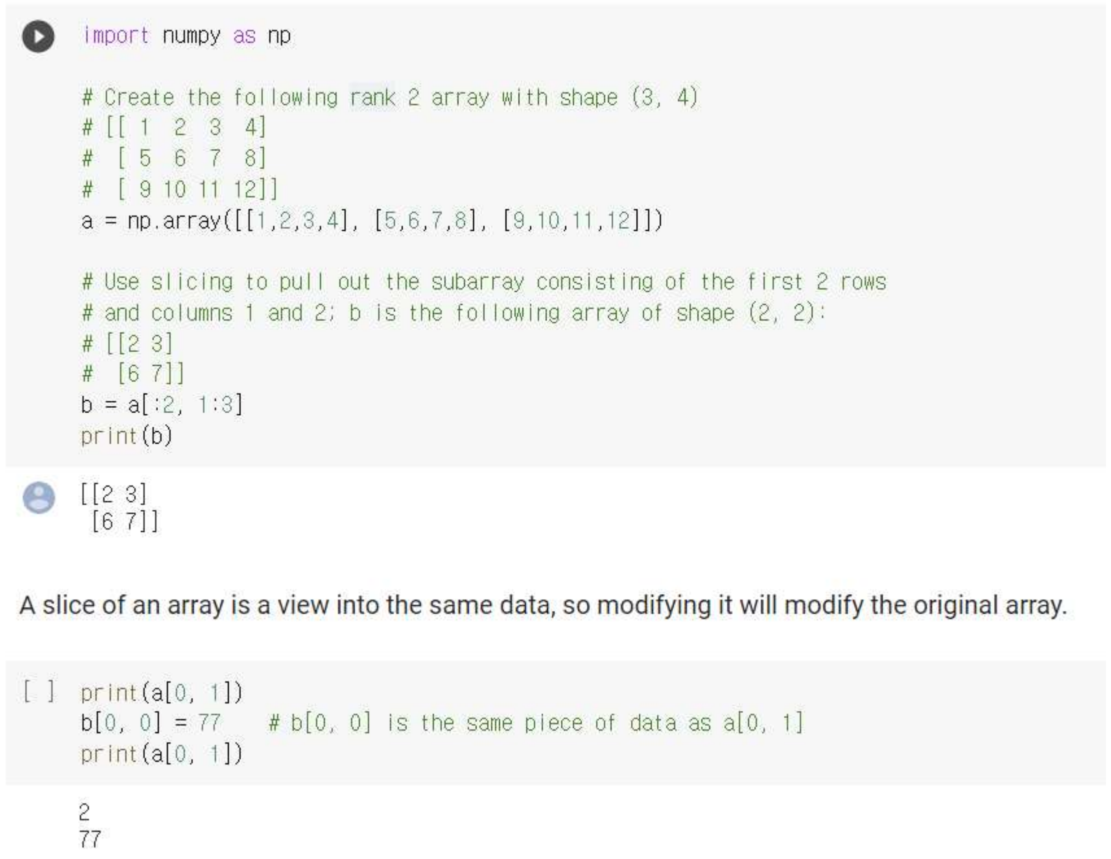


```
e = np.random.random((2,2)) # Create an array filled with random values
print(e)
```

```
[[0.8690054  0.57244319]
 [0.29647245 0.81464494]]
```

Numpy: Array indexing

- Slicing: Similar to Python lists, numpy arrays can be sliced.
- Since arrays may be multidimensional, you must specify a slice for each dimension of the array:



```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
print(b)
```

```
[[2 3]
 [6 7]]
```

A slice of an array is a view into the same data, so modifying it will modify the original array.

```
print(a[0, 1])
b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])
```

```
2
77
```

Numpy: Array indexing

- **Two ways of accessing the data in the middle row of the array.**
- **Mixing integer indexing with slices yields an array of lower rank, while using only slices yields an array of the same rank as the original array:**

```
[ ]  # Create the following rank 2 array with shape (3, 4)
     a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
     print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

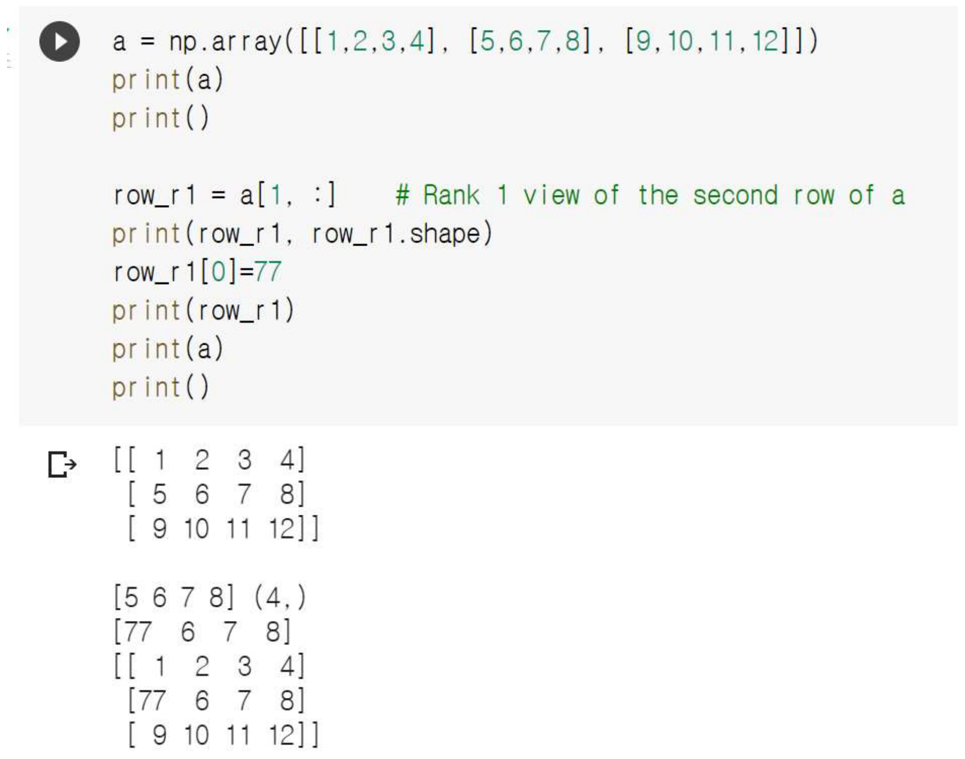
```
[ ]  row_r1 = a[1, :]    # Rank 1 view of the second row of a
     row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
     row_r3 = a[[1], :]  # Rank 2 view of the second row of a
     print(row_r1, row_r1.shape)
     print(row_r2, row_r2.shape)
     print(row_r3, row_r3.shape)
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[[5 6 7 8]] (1, 4)
```

```
[ ]  # We can make the same distinction when accessing columns of an array:
     col_r1 = a[:, 1]
     col_r2 = a[:, 1:2]
     print(col_r1, col_r1.shape)
     print()
     print(col_r2, col_r2.shape)
```

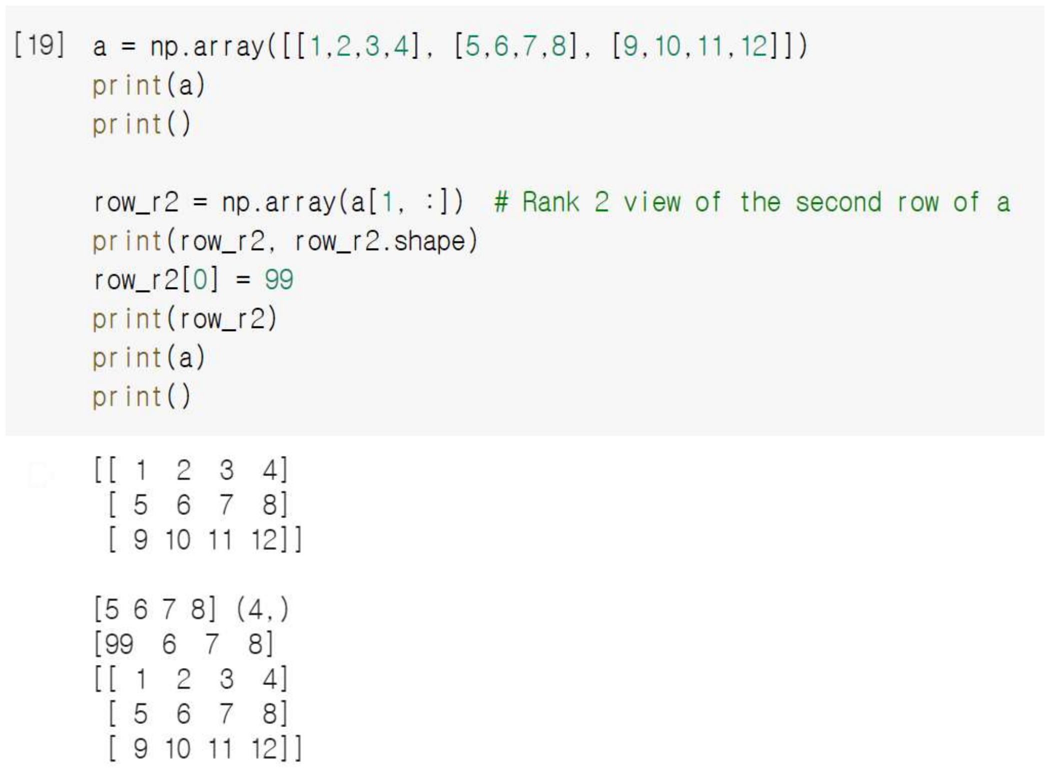
```
[ 2  6 10] (3,)

[[ 2]
 [ 6]
 [10]] (3, 1)
```

Numpy: Array indexing



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

row_r1 = a[1, :]    # Rank 1 view of the second row of a
print(row_r1, row_r1.shape)
row_r1[0]=77
print(row_r1)
print(a)
print()
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

[5 6 7 8] (4,)
[77  6  7  8]
[[ 1  2  3  4]
 [77  6  7  8]
 [ 9 10 11 12]]
```

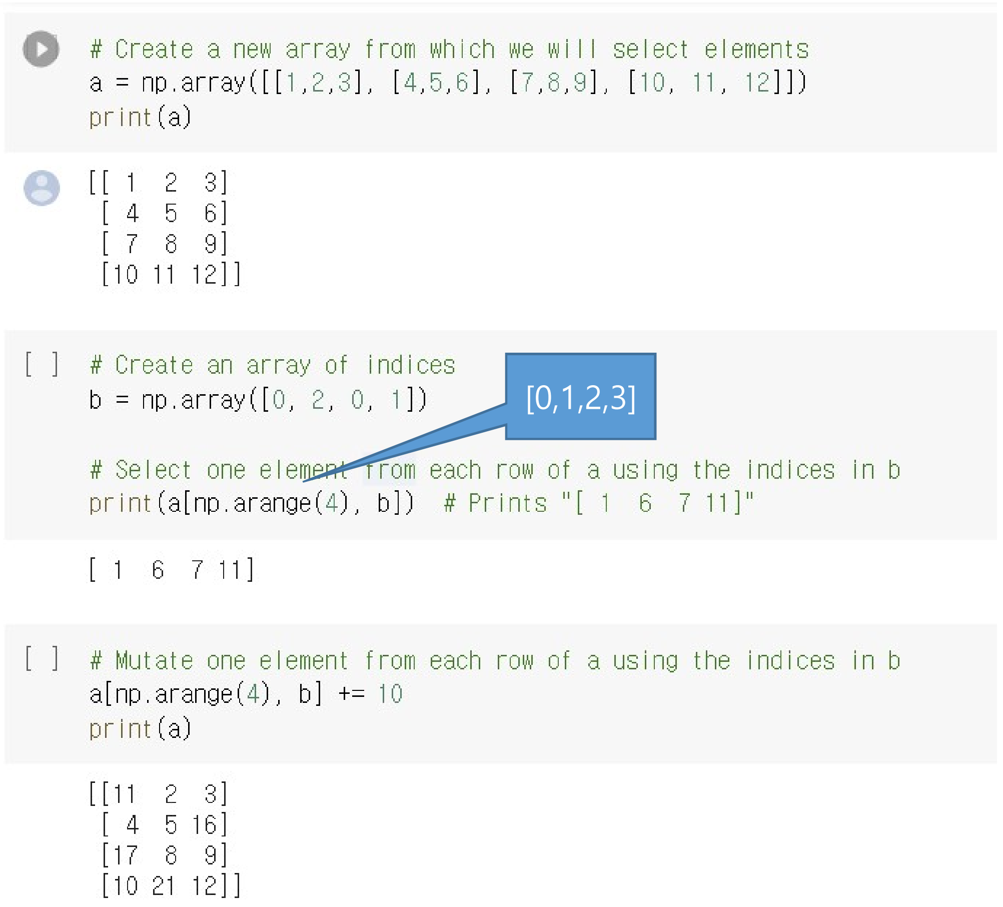


```
[19]  a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
      print(a)
      print()

      row_r2 = np.array(a[1, :])  # Rank 2 view of the second row of a
      print(row_r2, row_r2.shape)
      row_r2[0] = 99
      print(row_r2)
      print(a)
      print()
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

[5 6 7 8] (4,)
[99  6  7  8]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Numpy: Array indexing

- **One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:**



```
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

```
[ ]  # Create an array of indices
     b = np.array([0, 2, 0, 1])

     # Select one element from each row of a using the indices in b
     print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
```

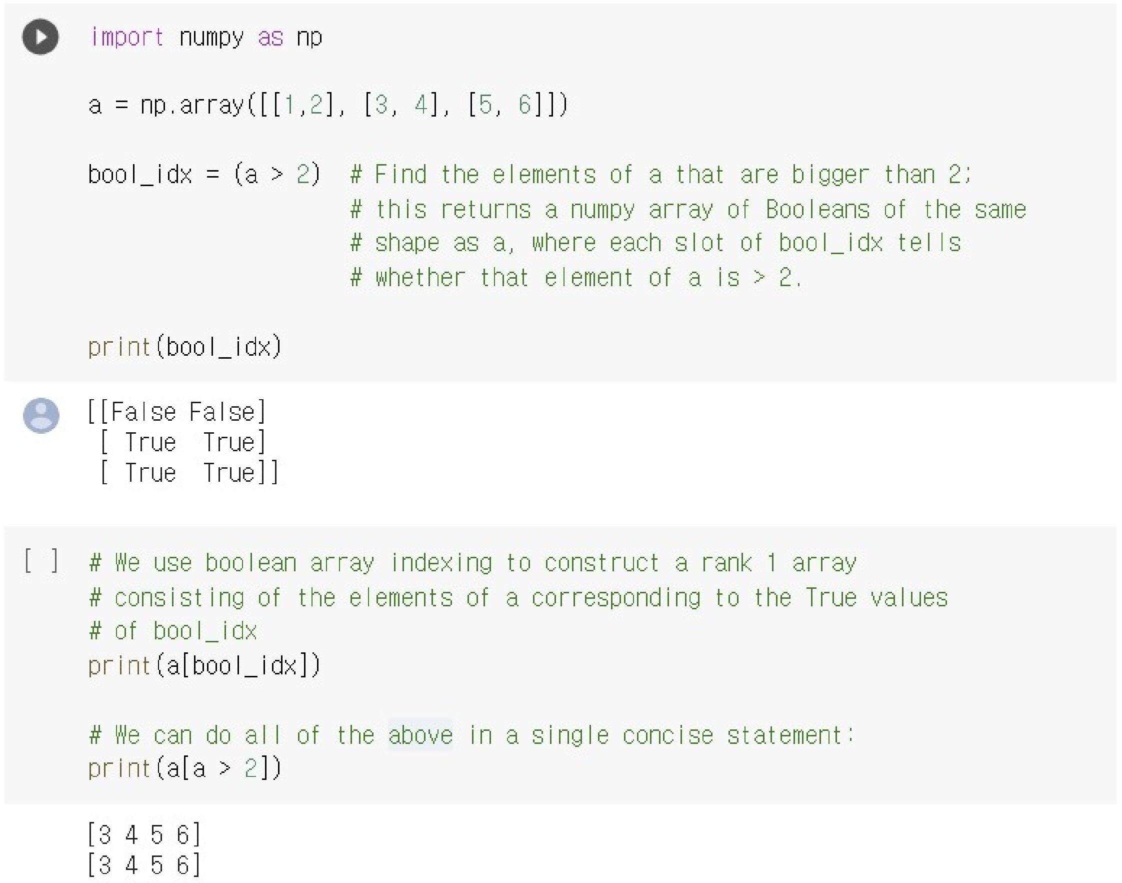
```
[ 1  6  7 11]
```

```
[ ]  # Mutate one element from each row of a using the indices in b
     a[np.arange(4), b] += 10
     print(a)
```

```
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Numpy: Array indexing

- **Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array.**



```
import numpy as np

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

bool_idx = (a > 2)  # Find the elements of a that are bigger than 2;
                    # this returns a numpy array of Booleans of the same
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.

print(bool_idx)
```

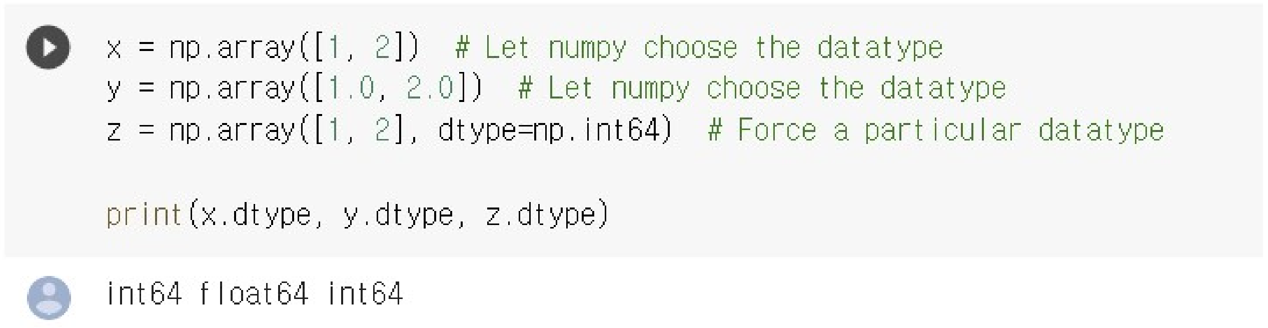
```
[[False False]
 [ True  True]
 [ True  True]]
```

```
[ ]  # We use boolean array indexing to construct a rank 1 array
     # consisting of the elements of a corresponding to the True values
     # of bool_idx
     print(a[bool_idx])

     # We can do all of the above in a single concise statement:
     print(a[a > 2])
```

```
[3 4 5 6]
[3 4 5 6]
```

Numpy: Datatypes

- **Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype.**



```python
x = np.array([1, 2])  # Let numpy choose the datatype
y = np.array([1.0, 2.0])  # Let numpy choose the datatype
z = np.array([1, 2], dtype=np.int64)  # Force a particular datatype

print(x.dtype, y.dtype, z.dtype)
```

```
int64 float64 int64
```

Numpy: Array math

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

```
[ ]  # Elementwise difference; both produce the array
     print(x - y)
     print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

```
[ ]  # Elementwise product; both produce the array
     print(x * y)
     print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

```
[ ]  # Elementwise division; both produce the array
     # [[ 0.2         0.33333333]
     #  [ 0.42857143  0.5       ]]
     print(x / y)
     print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

```
[ ]  # Elementwise square root; produces the array
     # [[ 1.          1.41421356]
     #  [ 1.73205081  2.        ]]
     print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Numpy: Array math

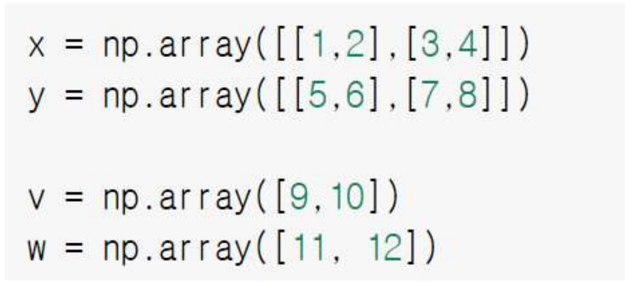
- We use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices.
- dot is available both as a function in the numpy module and as an instance method of array objects:

```
v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

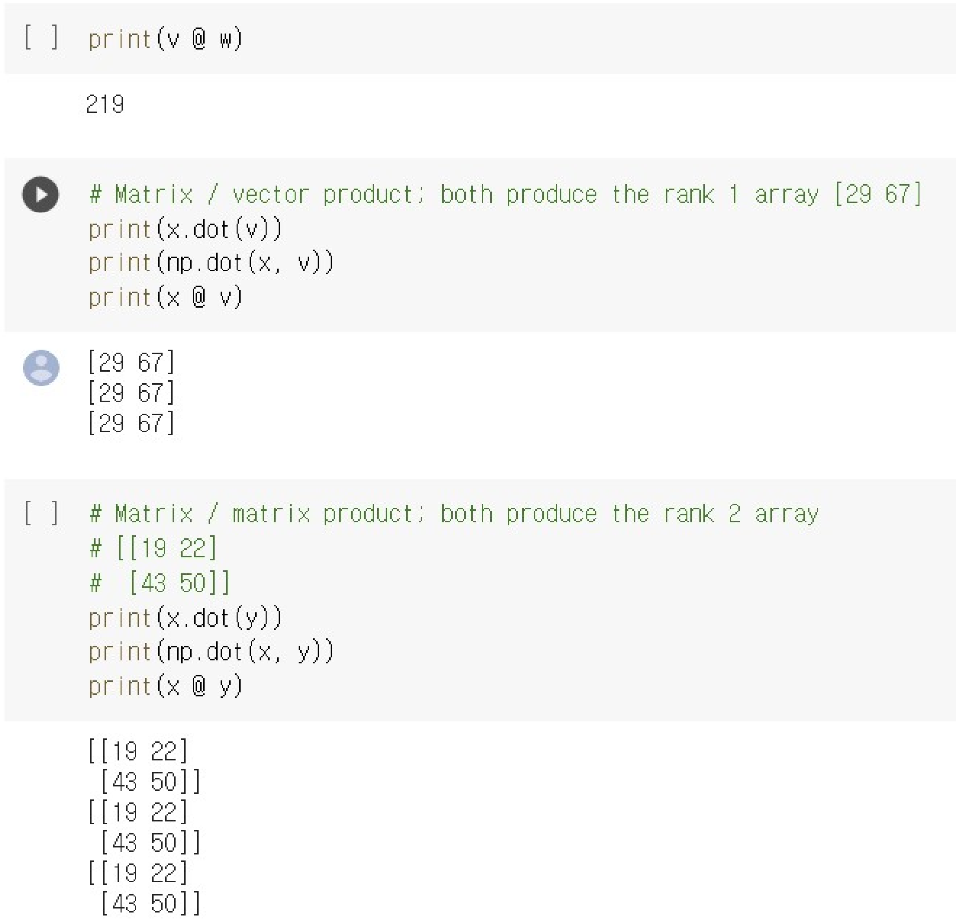
```
219
219
```

Numpy: Array math

- You can also use the `@` operator which is equivalent to numpy's `dot` operator.



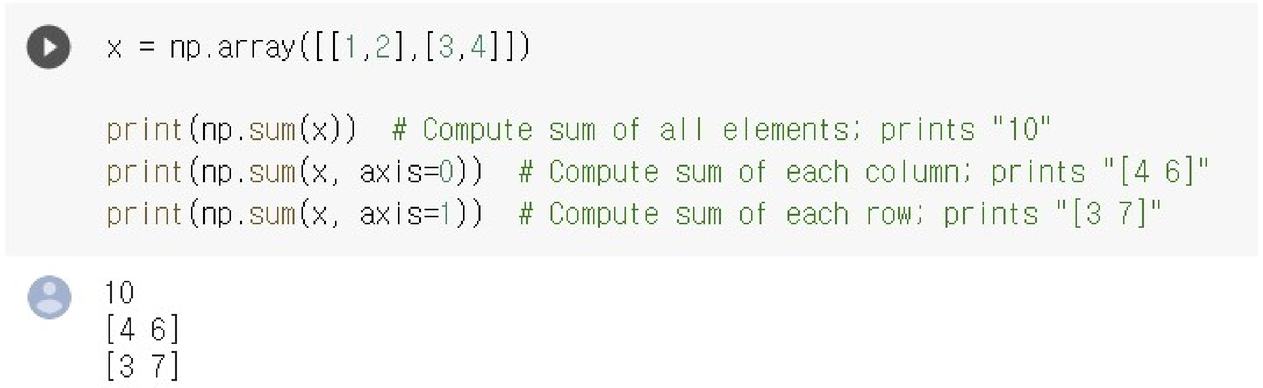
```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])
```



```
[ ]  print(v @ w)

     219

     # Matrix / vector product; both produce the rank 1 array [29 67]
     print(x.dot(v))
     print(np.dot(x, v))
     print(x @ v)

     [29 67]
     [29 67]
     [29 67]

[ ]  # Matrix / matrix product; both produce the rank 2 array
     # [[19 22]
     #  [43 50]]
     print(x.dot(y))
     print(np.dot(x, y))
     print(x @ y)

     [[19 22]
      [43 50]]
     [[19 22]
      [43 50]]
     [[19 22]
      [43 50]]
```

Numpy: Array math

- Numpy provides many useful functions for performing computations on arrays; one of the most useful is sum:

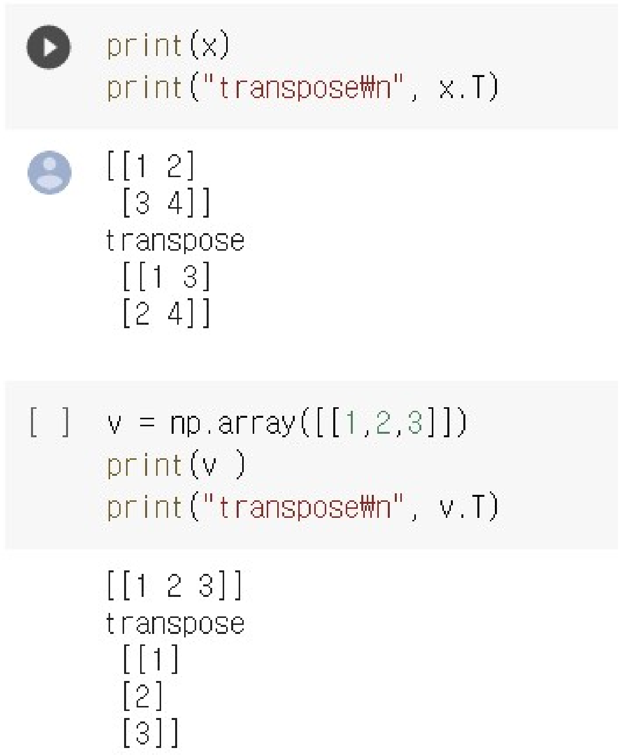


```
x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

Numpy: Array math

- **Transpose operation**



```
print(x)
print("transpose₩n", x.T)
```

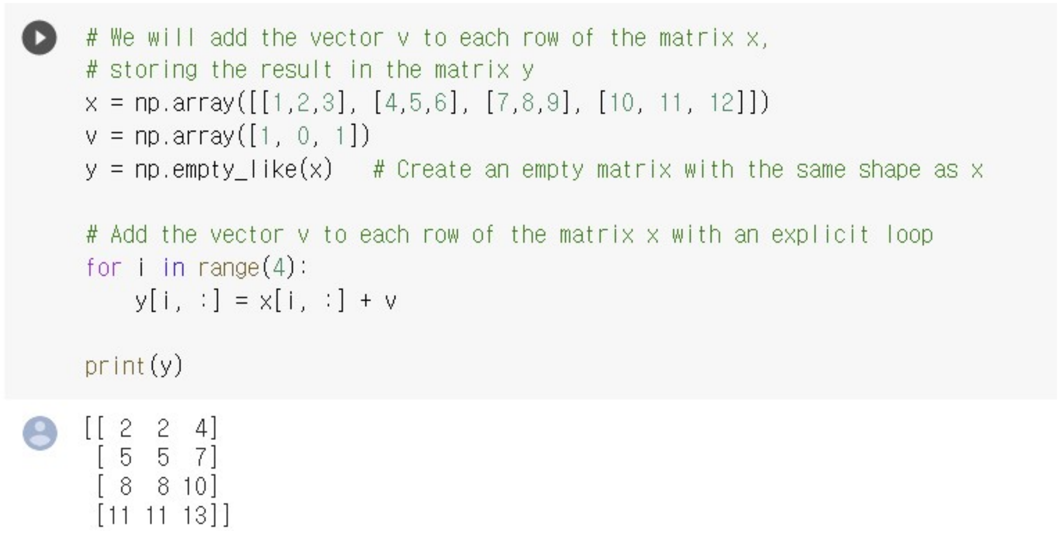
```
[[1 2]
 [3 4]]
transpose
 [[1 3]
 [2 4]]
```

```
v = np.array([[1,2,3]])
print(v )
print("transpose₩n", v.T)
```

```
[[1 2 3]]
transpose
 [[1]
 [2]
 [3]]
```

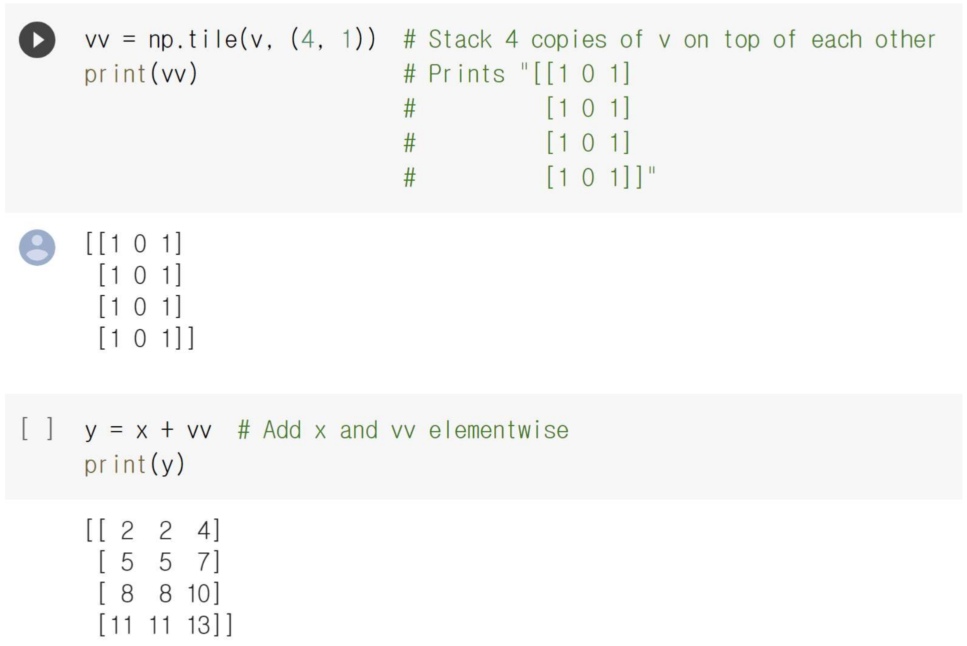
Numpy: Broadcasting

- **Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations.**



```
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)    # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```



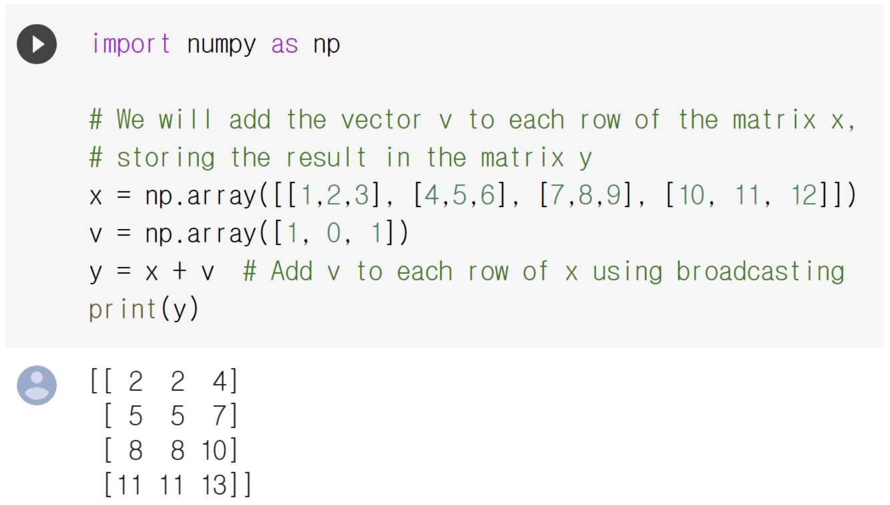
```
vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
print(vv)                # Prints "[[1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]
                         #          [1 0 1]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
```

```
[ ]  y = x + vv  # Add x and vv elementwise
     print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Numpy: Broadcasting



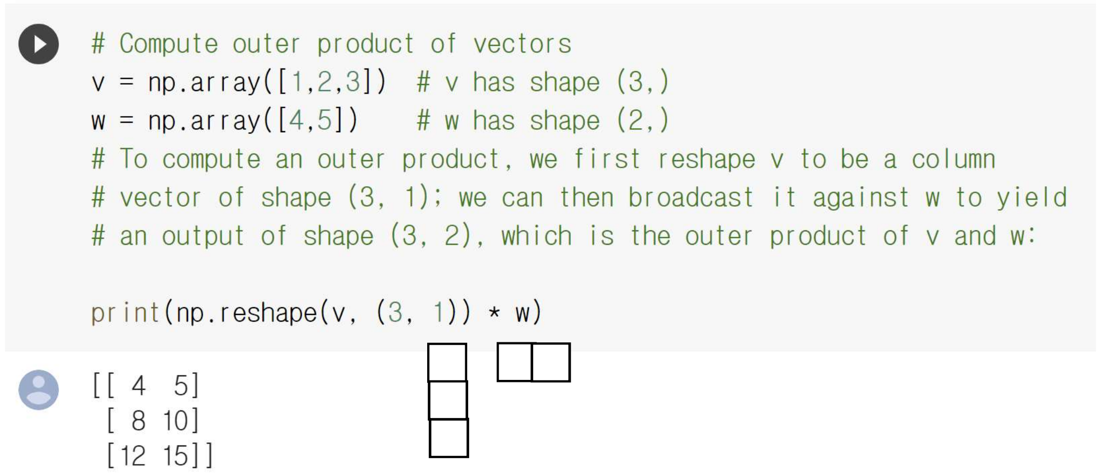
```python
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

- **The line y = x + v works even though x has shape (4, 3) and v has shape (3,) due to broadcasting; this line works as if v actually had shape (4, 3), where each row was a copy of v, and the sum was performed elementwise.**
- **Broadcasting two arrays together follows these rules:**
  - If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
  - The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
  - The arrays can be broadcast together if they are compatible in all dimensions.
  - After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
  - In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

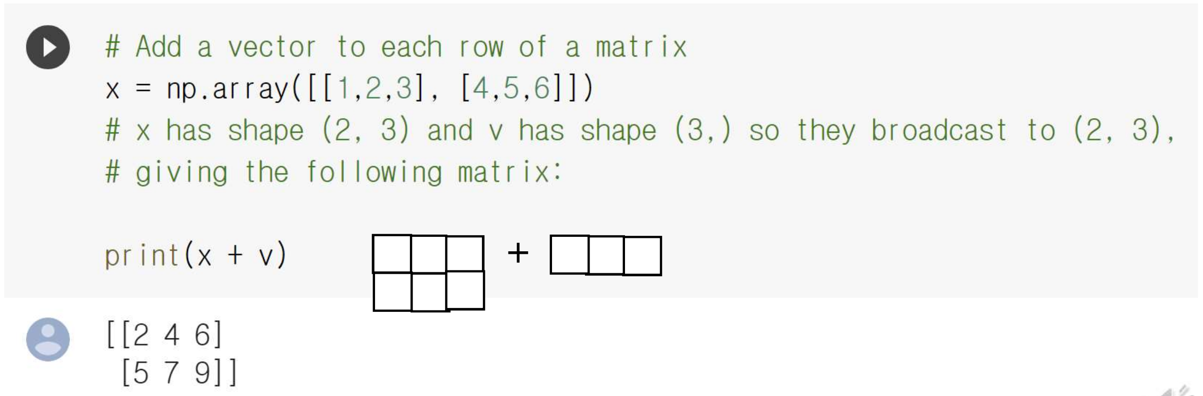
Numpy: Broadcasting

- **Here are some applications of broadcasting:**



```
# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:

print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

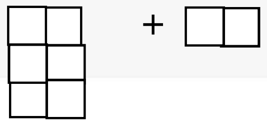


```
# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:

print(x + v)
```

```
[[2 4 6]
 [5 7 9]]
```

Numpy: Broadcasting

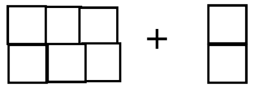
```
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:

print((x.T + w).T)
```



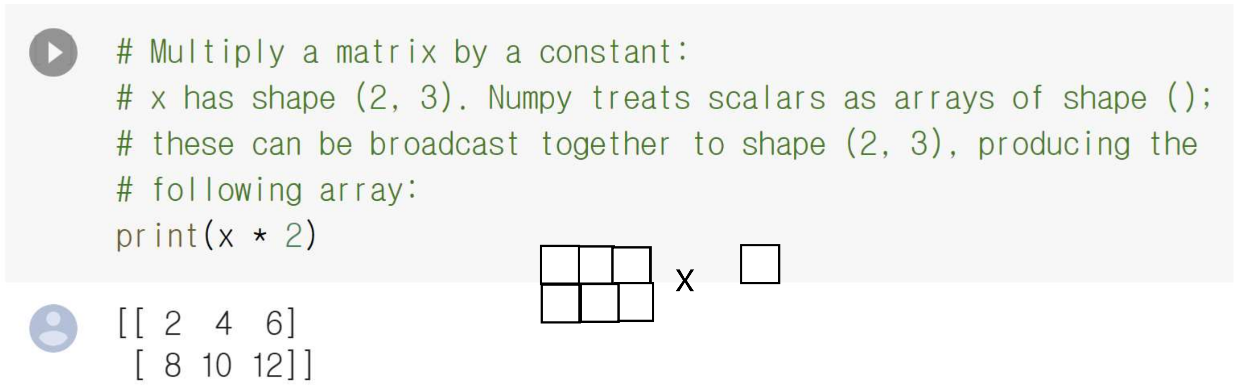
```
[[ 5  6  7]
 [ 9 10 11]]
```

```
[ ]  # Another solution is to reshape w to be a row vector of shape (2, 1);
     # we can then broadcast it directly against x to produce the same
     # output.
     print(x + np.reshape(w, (2, 1)))
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

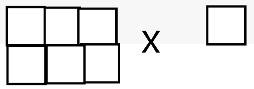


Numpy: Broadcasting



```
# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
print(x * 2)
```

```
[[ 2  4  6]
 [ 8 10 12]]
```



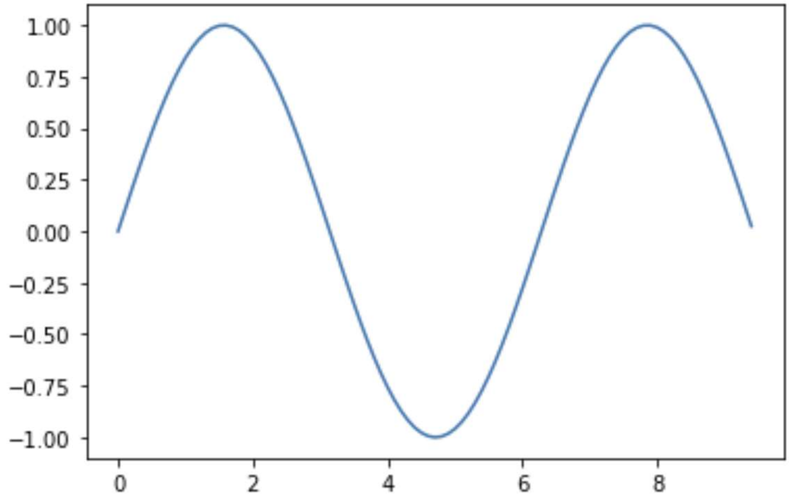
Matplotlib

- **Matplotlib is a plotting library**

```
[ ]  import matplotlib.pyplot as plt
```

```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
```

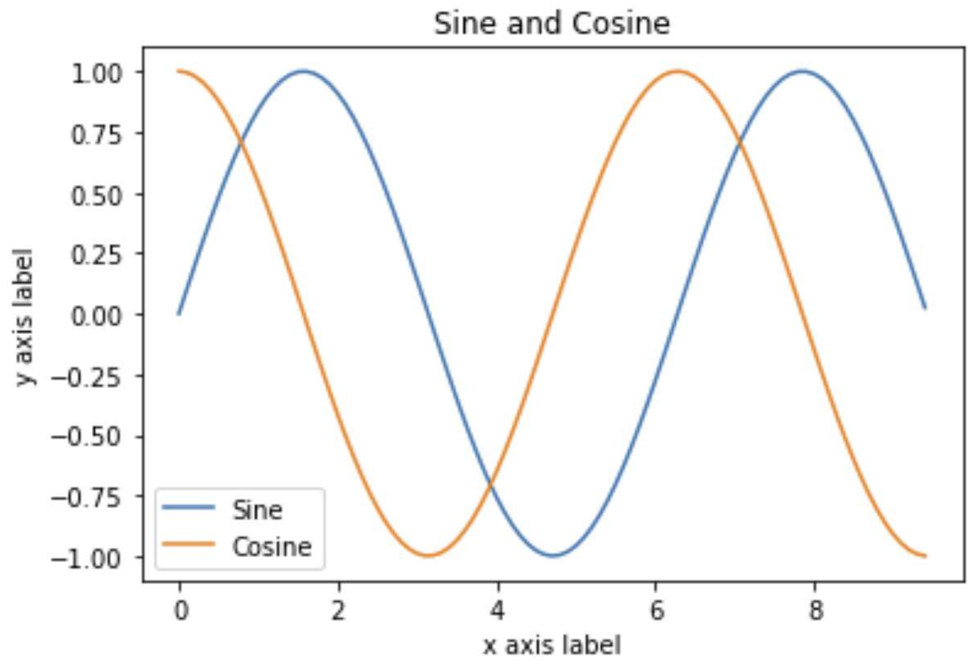
```
[<matplotlib.lines.Line2D at 0x7f0f3a0b4208>]
```



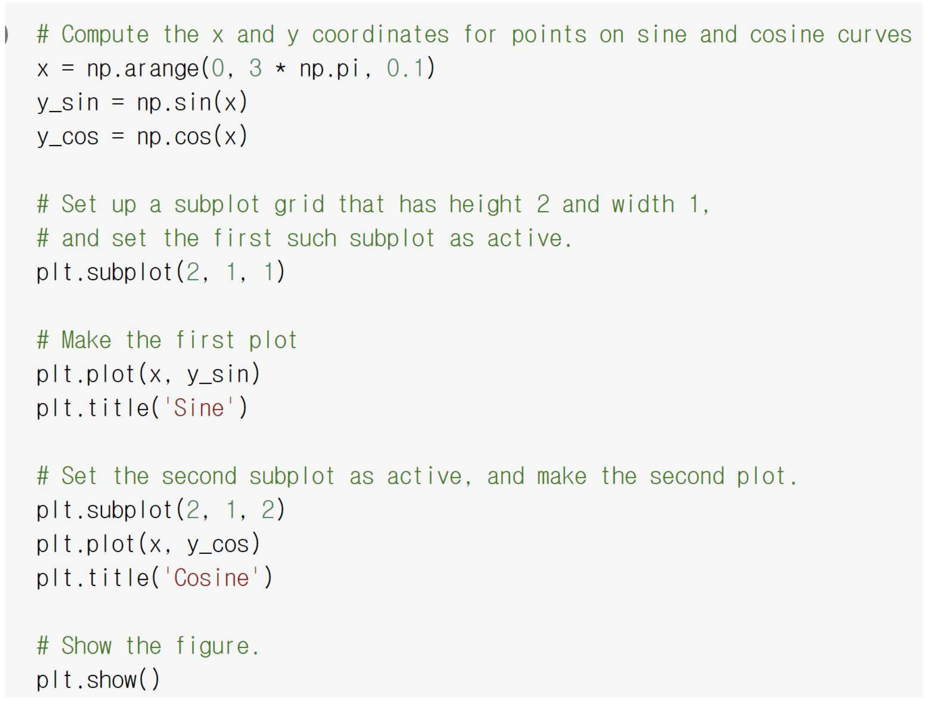
Matplotlib

```
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
```

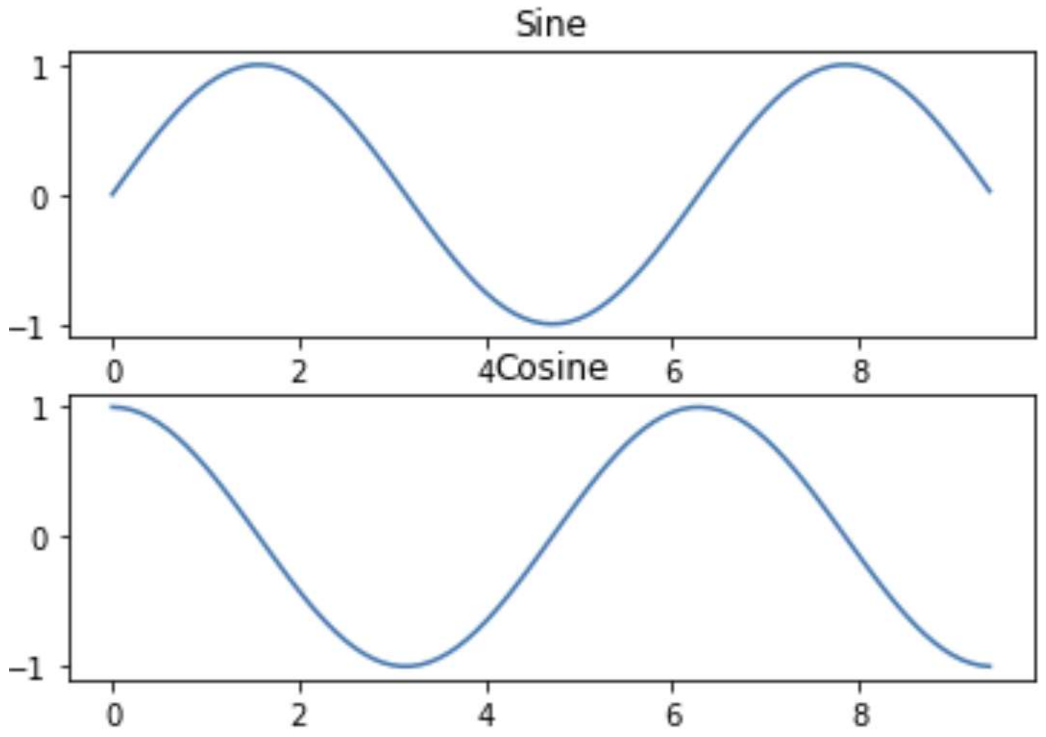
```
<matplotlib.legend.Legend at 0x7f0f39c04780>
```



Matplotlib



```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```


Sine
Cosine