

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

```
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 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]
```

You can find the full list of mathematical functions provided by numpy in the [documentation](#).

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the `T` attribute of an array object:

```
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]]
```

## ✓ Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

```
# 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]]
```

This works; however when the matrix `x` is very large, computing an explicit loop in Python could be slow. Note that adding the vector `v` to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `x` and `vv`. We could implement this approach like this:

```
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 allows us to perform this computation without actually creating multiple copies of `v`. Consider this version, using broadcasting:

```
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:

1. 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.
2. 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.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. 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

If this explanation does not make sense, try reading the explanation from the [documentation](#) or this [explanation](#).

Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the [documentation](#).

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]]
```

```
# 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]]
```

```
# 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]]
```

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.

This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the [numpy reference](#) to find out much more about numpy.

## ✓ Matplotlib

Matplotlib is a plotting library. In this section give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to that of MATLAB.

```
import matplotlib.pyplot as plt
```

By running this special iPython command, we will be displaying plots inline:


```
%matplotlib inline
```

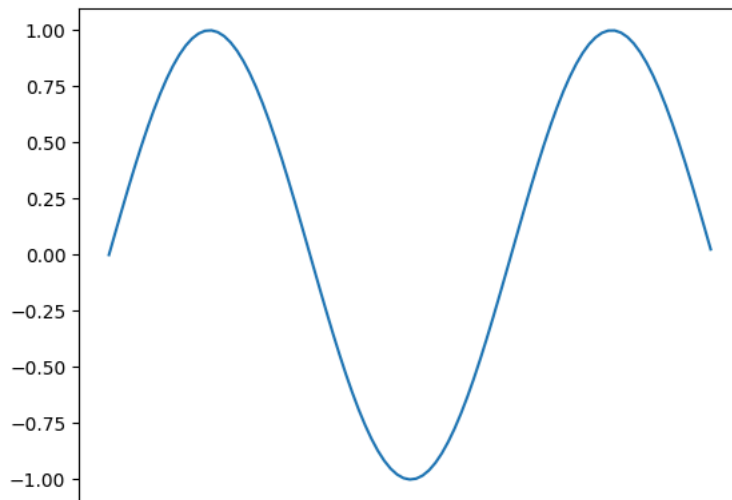
## ✓ Plotting

The most important function in `matplotlib` is `plot`, which allows you to plot 2D data. Here is a simple example:

```
# 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 `0x7dd64091bbb0`]

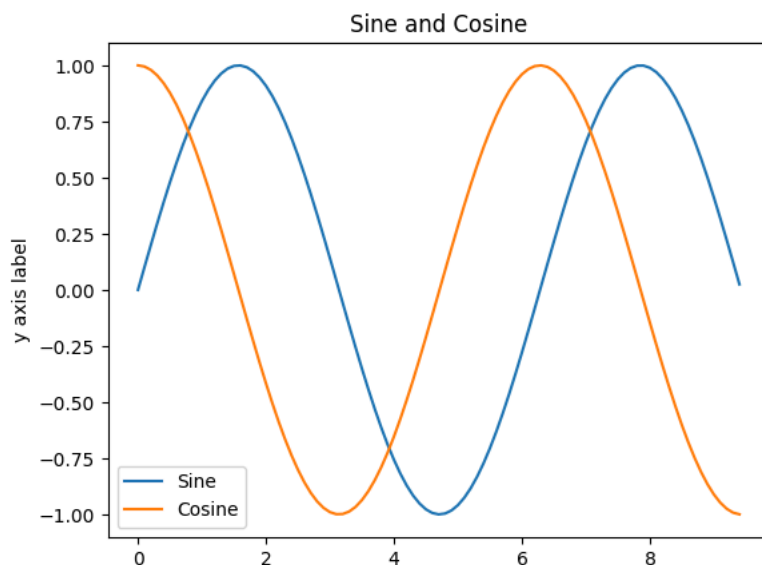


With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

```
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 `0x7dd6408c87f0`



## Subplots

You can plot different things in the same figure using the `subplot` function. Here is an example:

```
# 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.
```

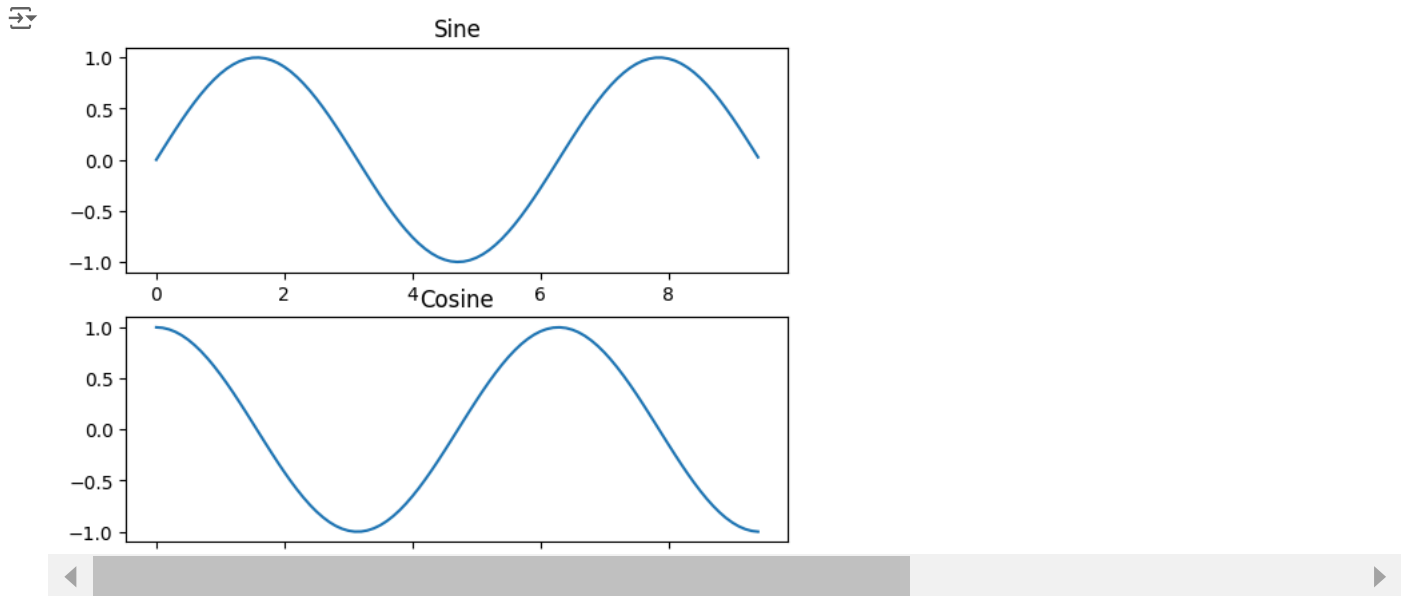
```
plt.subplot(2, 1, 2)
```

```
plt.plot(x, y_cos)
```

```
plt.title('Cosine')
```

```
# Show the figure.
```

```
plt.show()
```



You can read much more about the `subplot` function in the [documentation](#).

Start coding or [generate](#) with AI.