10 Learning & Programming Python

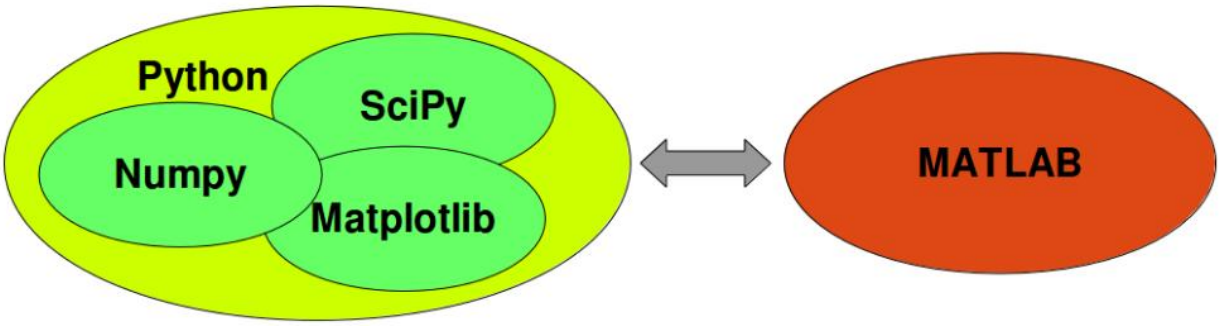
0.1 Objective

The objectives of the experiment is to learn the following:

- Give a quick introduction about Python vs. Matlab.
- Download libraries in Python.
- Show some examples about NumPy Library.
- Show Difference between NumPy Arrays and Python standard lists.
- Give some examples about SciPy Library.
- Show how to plot functions in Python using Matplotlib Library.

0.2 The Python Alternative to Matlab

Python in combination with Numpy, Scipy and Matplotlib can be used as a replacement for MATLAB. The combination of NumPy, SciPy and Matplotlib is a free alternative to MATLAB. Even though MATLAB has a huge number of additional toolboxes available, NumPy has the advantage that Python is a more modern and complete programming language and - as we have said already before - is open source. SciPy adds even more MATLAB-like functionalities to Python. Python is rounded out in the direction of MATLAB with the module Matplotlib, which provides MATLAB-like plotting functionality.



0.3 Download Libraries(Modules) in Python

The standard packaging tools are all designed to be used from the command line.
The following command will install the latest version of a module and its dependencies from the Python Packaging Index:

```
python -m pip install SomePackage
```

Example:

```
python -m pip install numpy
```

0.4 NumPy Library

Numpy is the core library for scientific computing in Python. It is an open source extension module for Python, which provides fast-precompiled functions for mathematical and numerical routines. It is enriches the programming language Python with powerful data structures for efficient computation of multi-dimensional arrays and matrices. The implementation is even aiming at huge matrices and arrays. Besides that, the module supplies a large library of high-level mathematical functions to operate on these matrices and arrays.

0.4.1 Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the *rank* of the array; the *shape* of an array is a tuple of integers giving the size of the array along each dimension.

Example:

```
import numpy as np
a = np.array([1, 2, 3])          # Create a rank 1 array
print(type(a))                   # Prints "<type 'numpy.ndarray'>"
print(a.shape)                   # Prints "(3,)"
print(a[0], a[1], a[2])          # Prints "1 2 3"
a[0] = 5                         # Change an element of the array
print(a)                         # Prints "[5, 2, 3]"
b = np.array([[1,2,3],[4,5,6]])  # Create a rank 2 array
print(b.shape)                   # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0]) # Prints "1 2 4"
```

Numpy also provides many functions to create arrays:

Example:

```
import numpy as np
a = np.zeros((2,2))       # Create an array of all zeros
print(a)                  # Prints "[[ 0.  0.]
                          #         [ 0.  0.]]"
b = np.ones((1,2))        # Create an array of all ones
print(b)                  # Prints "[[ 1.  1.]]"
c = np.full((2,2), 7,dtype="int64")      # Create a constant array
print(c)                  # Prints "[[ 7.  7.]
                          #         [ 7.  7.]]"
d = np.eye(2)             # Create a 2x2 identity matrix
print(d)                  # Prints "[[ 1.  0.]
                          #         [ 0.  1.]]"
e=np.random.random((2,2)) #Create an array filled with random value
print(e)                  # Might print "[[ 0.9194017 0.0843941]
```

0.4.2 Array indexing

Numpy offers several ways to index into arrays.

A) 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
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
b = a[:2, 1:3]
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array. Note that this is quite different from the way that MATLAB handles array slicing:

```
import numpy as np
a = np.array([[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[1:3, :]        # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #         [ 6]
                             #         [10]] (3, 1)"
```

B) Integer array indexing:

When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
print(a[[0, 1, 2], [0, 1, 0]])             # Prints "[1 4 5]"
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"
print(a[[0, 0], [1, 1]])                   # Prints "[2 2]"
print(np.array([a[0, 1], a[0, 1]]))        # Prints "[2 2]"
```

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

```
import numpy as np
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)  # prints "array([[ 1,  2,  3],[ 4,  5,  6],[ 7,  8,  9], [10, 11, 12]])"
b = np.array([0, 2, 0, 1])
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
a[np.arange(4), b] += 10
print(a)  # prints "array([[11,  2,  3],
          #              [ 4,  5, 16],
          #              [17,  8,  9],
          #              [10, 21, 12]])
```

C) Boolean array indexing:

Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. **Example**:

```
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
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.
print(bool_idx)     # Prints "[[False False]
                    #         [ True  True]
                    #         [ True  True]]"
print(a[bool_idx])  # Prints "[3 4 5 6]"
print(a[a > 2])     # Prints "[3 4 5 6]"
```

0.4.3 Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. 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.

Example:

```
import numpy as np
x = np.array([1, 2])                # Let numpy choose the datatype
print (x.dtype)                     # Prints "int64"
x = np.array([1.0, 2.0])            # Let numpy choose the datatype
print (x.dtype)                     # Prints "float64"
x = np.array([1, 2], dtype=np.int64)  # Force a particular datatype
print (x.dtype)                     # Prints "int64"
```

0.4.4 Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
import numpy as np
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
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))
# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
```

```
# Elementwise product; both produce the array
                # [[ 5.0 12.0]
                #  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))
# Elementwise division; both produce the array
# [[ 0.2        0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
# Elementwise square root; produces the array
# [[ 1.         1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

We can 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:

Example:

```
import numpy as np
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.dot(w))
print(np.dot(v, w))
print(x.dot(v))
print(np.dot(x, v))
print(x.dot(y))
print(np.dot(x, y))
```

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

```
import numpy as np
x = np.array([[1,2],[3,4]])
print(np.sum(x))
print(np.sum(x, axis=0))
print(np.sum(x, axis=1))
```

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:

```
import numpy as np
x = np.array([[1,2], [3,4]])
print(x)
print(x.T)
```

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

0.4.5 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:

```
import numpy as np
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 empty matrix with the same shape as x
for i in range(4):
   y[i, :] = x[i, :] + v
print(y)

x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
print(vv)

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

0.4.6 Creation of Evenly Spaced Values

There are functions provided by Numpy to create evenly spaced values within a given interval. One uses a given distance 'arange' and the other one 'linspace' needs the number of elements and creates the distance automatically.

Arange

The syntax of arange:

```
arange([start], stop,[step], dtype=None)
```

Example:

```
import numpy as np
a = np.arange(1, 10)
print(a)
# compare to range:
x = range(1,10)
print(x)    # x is an iterator
print(list(x))
```

```
# some more arange examples:
x = np.arange(10.4)
print(x)
x = np.arange(0.5, 10.4, 0.8)
print(x)
x = np.arange(0.5, 10.4, 0.8, int)
print(x)
```

Linspace

The syntax of linspace:

linspace(start, stop, num=50, endpoint=True, retstep=False)

Example:

```
import numpy as np
# 50 values between 1 and 10:
print(np.linspace(1, 10))
# 7 values between 1 and 10:
print(np.linspace(1, 10, 7))
# excluding the endpoint:
print(np.linspace(1, 10, 7, endpoint=False))
```

Time Comparison between Python Lists and Numpy Arrays

One of the main advantages of NumPy is its advantage in **time compared** to standard Python. Let's look at the following functions:

```
import time
import numpy as np
size_of_vec = 1000
def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = []
    for i in range(len(X)):
        Z.append(X[i] + Y[i])
    return time.time() - t1
def numpy_version():
    t1 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    return time.time() - t1
t1 = pure_python_version()
t2 = numpy_version()
print(t1, t2)
print("Numpy is in this example " + str(t1/t2) + " faster!")
```

This gets us the following:

```
0.0002090930938720703 2.0503997802734375e-05
```

Numpy is in this example 10.19767441860465 faster!

0.5 SciPy Library

The scipy library contains various toolboxes dedicated to common issues in scientific computing. Its different submodules correspond to different applications, such as interpolation, integration, optimization, image processing, statistics, special functions, etc.

SciPy is composed of task-specific sub-modules:

scipy.cluster	Vector quantization / Kmeans
scipy.constants	Physical and mathematical constants
scipy.fftpack	Fourier transform
scipy.integrate	Integration routines
scipy.interpolate	Interpolation
scipy.io	Data input and output
scipy.linalg	Linear algebra routines
scipy.ndimage	n-dimensional image package
scipy.odr	Orthogonal distance regression
scipy.optimize	Optimization
scipy.signal	Signal processing
scipy.sparse	Sparse matrices
scipy.spatial	Spatial data structures and algorithms
scipy.special	Any special mathematical functions
scipy.stats	Statistics

Linear algebra operations: scipy.linalg

The **scipy.linalg.det()** function computes the determinant of a square matrix:

```
import numpy as np
from scipy import linalg
arr = np.array([[1, 2],[3, 4]])
print(linalg.det(arr))
arr = np.array([[3, 2],[6, 4]])
print(linalg.det(arr))
```

The **scipy.linalg.inv()** function computes the inverse of a square matrix:

```
import numpy as np
from scipy import linalg
arr = np.array([[1, 2],[3, 4]])
iarr = linalg.inv(arr)
print(iarr)
```

Fast Fourier transforms: scipy.fftpack

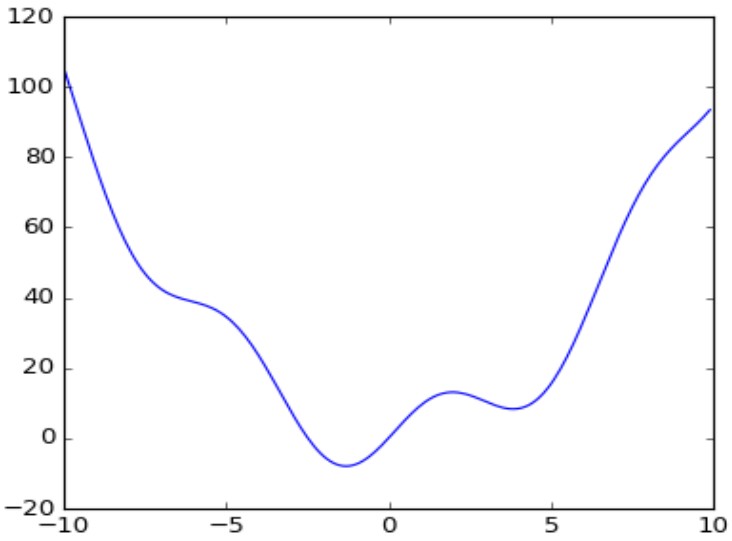
```
from scipy import fftpack
import numpy as  np
time_step = 0.02
period = 5
time_vec = np.arange(0, 20, time_step)
sig = np.sin(2 * np.pi / period * time_vec) + 0.5 * np.random.randn(time_vec.size)
sample_freq = fftpack.fftfreq(sig.size, d=time_step)
```

```
print(fftpack.fft(sig))
```

Optimization and fit: scipy.optimize

Optimization is the problem of finding a numerical solution to a minimization or equality.
The scipy.optimize module provides useful algorithms for function minimization (scalar or multi-dimensional), curve fitting and root finding.

```
from scipy import optimize
from numpy import np
import matplotlib.pyplot as plt
def f(x):
        return x**2 + 10*np.sin(x)
x = np.arange(-10, 10, 0.1)
```



This function has a global minimum around -1.3 and a local minimum around 3.8.
To find the local minimum, let's constraint the variable to the interval (0, 10) using **scipy.optimize.fminbound():**

```
xmin_local = optimize.fminbound(f, 0, 10)
print(xmin_local)
```

Finding the roots of a scalar function

To find a root, i.e. a point where $f(x) = 0$, of the function f above we can use for example **scipy.optimize.fsolve()**:

```
root = optimize.fsolve(f, 1)  # our initial guess is 1
print(root)
```

Note that only one root is found. Inspecting the plot of f reveals that there is a second root around -2.5. We find the exact value of it by adjusting our initial guess:

```
root2 = optimize.fsolve(f, -2.5)
print(root2)
```

Curve fitting

Suppose we have data sampled from f with some noise:

```
xdata = np.linspace(-10, 10, num=20)
ydata = f(xdata) + np.random.randn(xdata.size)
```

Now if we know the functional form of the function from which the samples were drawn ($x^2 + \sin(x)$ in this case) but not the amplitudes of the terms, we can find those by least squares curve fitting. First we have to define the function to fit:

```
def f2(x, a, b):
        return a*x**2 + b*np.sin(x)
```

Then we can use **scipy.optimize.curve_fit()** to find a and b:

```
guess = [2, 2]
params, params_covariance = optimize.curve_fit(f2, xdata, ydata, guess)
print(params)
```

Interpolation: scipy.interpolate

The **scipy.interpolate.interp1d** class can build a linear interpolation function:

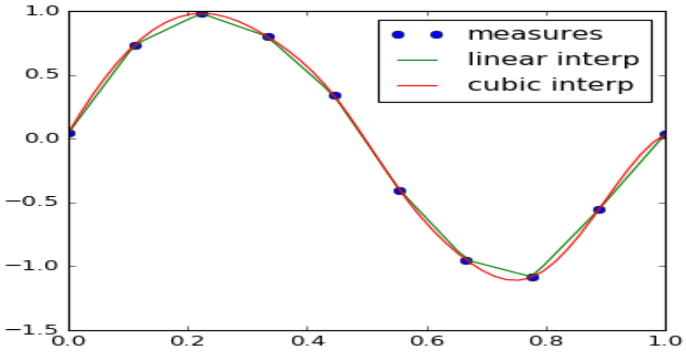
```
from scipy.interpolate import interp1d
import numpy as np
measured_time = np.linspace(0, 1, 10)
noise = (np.random.random(10)*2 - 1) * 1e-1
measures = np.sin(2 * np.pi * measured_time) + noise
linear_interp = interp1d(measured_time, measures)
```

Then the **scipy.interpolate.linear_interp** instance needs to be evaluated at the time of interest:

```
computed_time = np.linspace(0, 1, 50)
linear_results = linear_interp(computed_time)
print(linear_results)
```

A cubic interpolation can also be selected by providing the kind optional keyword argument:

```
cubic_interp  =interp1d(measured_time,measures,kind='cubic')
cubic_results = cubic_interp(computed_time)
print(cubic_results)
```

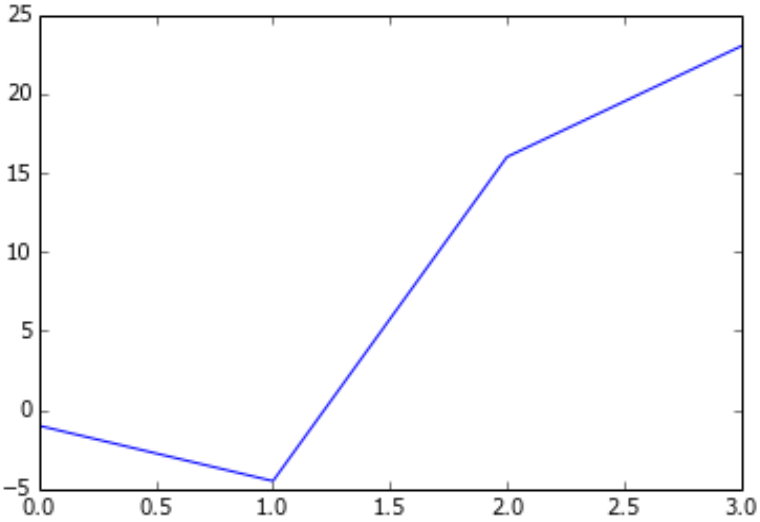


0.6 Matplotlib Library

Matplotlib is probably the single most used Python package for 2D-graphics. It provides both a very quick way to visualize data from Python and publication-quality figures in many formats. We are going to explore matplotlib in interactive mode covering most common cases.

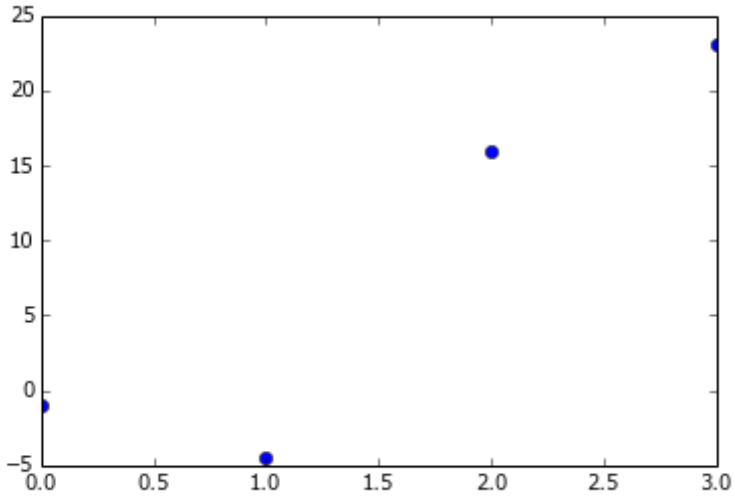
We will start with a simple graph. Is is common practice to rename matplotlib.pyplot to plt. We will use the plot function of pyplot in our first example. We will pass a list of values to the plot function. Plot takes these as Y values. The indices of the list are automatically taken as the X values.

```
import matplotlib.pyplot as plt
plt.plot([-1, -4.5, 16, 23])
plt.show()
```



What we see is a continuous graph, even though we provided discrete data for the Y values. By adding a format string to the function call of plot, we can create a graph with discrete values, in our case blue circle markers. The format string defines the way how the discrete points have to be rendered.

```
import matplotlib.pyplot as plt
plt.plot([-1, -4.5, 16, 23], "ob")
plt.show()
```



The format parameter of pyplot.plot

We have used "ob" in our previous example as the format parameter. It consists of two characters. The first one defines the line style or the dicrete value style, i.e. the markers, while the second one chooses a colour for the graph. The order of the two characters could have been reversed, i.e. we could have written it as "bo" as well. If the format parameter is not giving, as it has been the case in our first example, "b-" is used as the default value, i.e. a solid blue line.

The following format string characters are accepted to control the line style or marker:

```
============================================
character       description
============================================
'-'             solid line style
'--'            dashed line style
'-.'            dash-dot line style
':'             dotted line style
'.'             point marker
','             pixel marker
'o'             circle marker
'v'             triangle_down marker
'^'             triangle_up marker
'<'             triangle_left marker
'>'             triangle_right marker
'1'             tri_down marker
'2'             tri_up marker
'3'             tri_left marker
'4'             tri_right marker
's'             square marker
'p'             pentagon marker
'*'             star marker
'h'             hexagon1 marker
'H'             hexagon2 marker
'+'             plus marker
'x'             x marker
'D'             diamond marker
'd'             thin_diamond marker
'|'             vline marker
'_'             hline marker
============================================
```

The following color abbreviations are supported:

```
==================
character   color
==================
'b'         blue
'g'         green
'r'         red
'c'         cyan
'm'         magenta
'y'         yellow
'k'         black
'w'         white
==================
```

As you may have guessed already, you can X values to the plot function. We will pass the multiples of 3 up 21 including the 0 to plot in the following example:

```
import matplotlib.pyplot as plt
# our X values:
days = list(range(0, 22, 3))
print(days)
# our Y values:
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
plt.plot(days, celsius values)
plt.show()
```

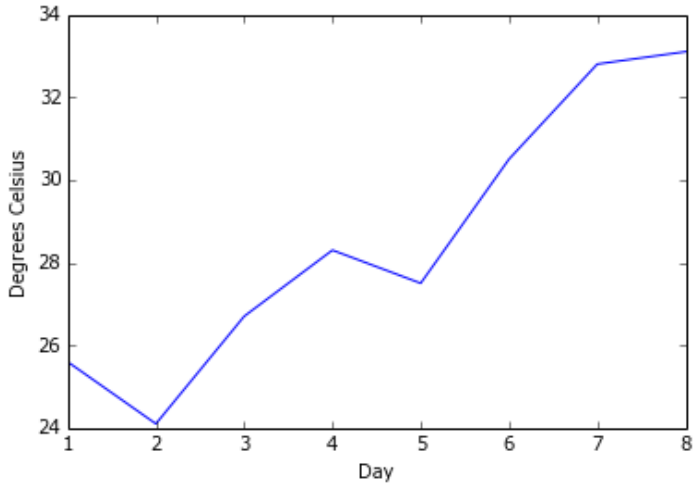
and once more with discrete values:

```
plt.plot(days, celsius_values, 'bo')
plt.show()
```

Labels on Axes

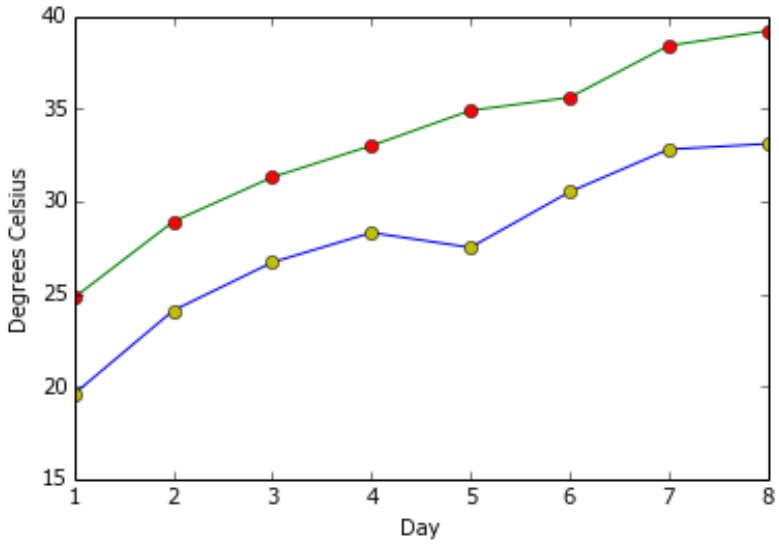
We can improve the appearance of our graph by adding labels to the axes. This can be done with the ylabel and xlabel function of pyplot.

```
import matplotlib.pyplot as plt
days = list(range(1,9))
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
plt.plot(days, celsius_values)
plt.xlabel('Day')
plt.ylabel('Degrees Celsius')
plt.show()
```



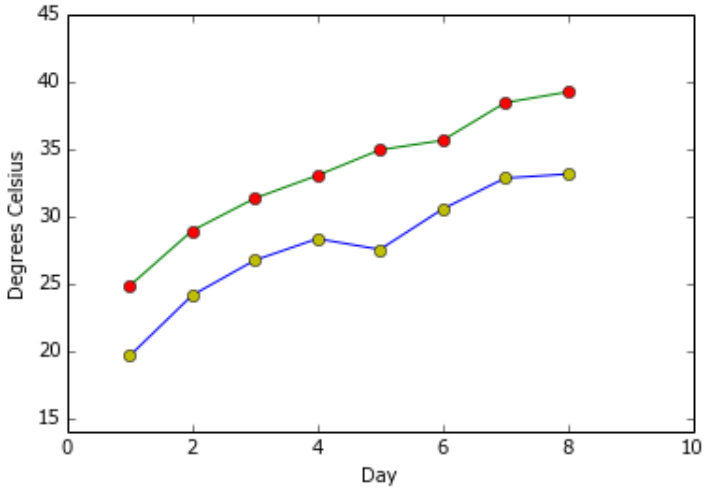
We can specify an arbitrary number of x, y, fmt groups in a plot function. In the following example, we use two different lists of y values:

```
import matplotlib.pyplot as plt
days = list(range(1,9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]
plt.xlabel('Day')
plt.ylabel('Degrees Celsius')
plt.plot(days, celsius_min,
         days, celsius_min, "oy",
         days, celsius_max,
         days, celsius_max, "or")
plt.show()
```



Checking and Defining the Range of Axes

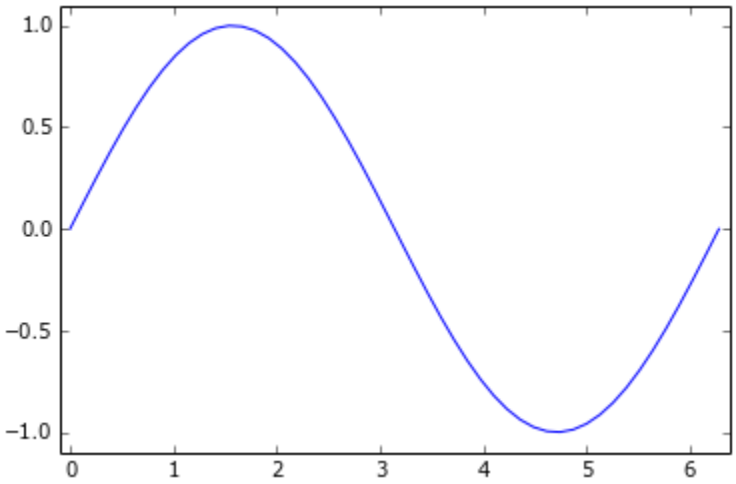
```
import matplotlib.pyplot as plt
days = list(range(1,9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]
plt.xlabel('Day')
plt.ylabel('Degrees Celsius')
plt.plot(days, celsius_min,
         days, celsius_min, "oy",
         days, celsius_max,
         days, celsius_max, "or")
print("The current limits for the axes are:")
print(plt.axis())
print("We set the axes to the following values:")
xmin, xmax, ymin, ymax = 0, 10, 14, 45
print(xmin, xmax, ymin, ymax)
plt.axis([xmin, xmax, ymin, ymax])
plt.show()
```



"linspace" to Define X Values

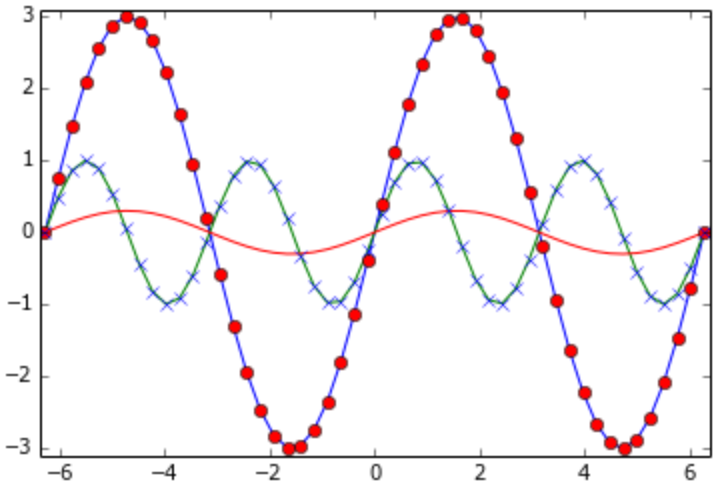
We will use the Numpy function linspace in the following example. linspace can be used to create evenly spaced numbers over a specified interval.

```
import numpy as np
import matplotlib.pyplot as plt
X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
F = np.sin(X)
plt.plot(X,F)
startx, endx = -0.1, 2*np.pi + 0.1
starty, endy = -1.1, 1.1
plt.axis([startx, endx, starty, endy])
plt.show()
```



We will just add two more plots with discrete points:

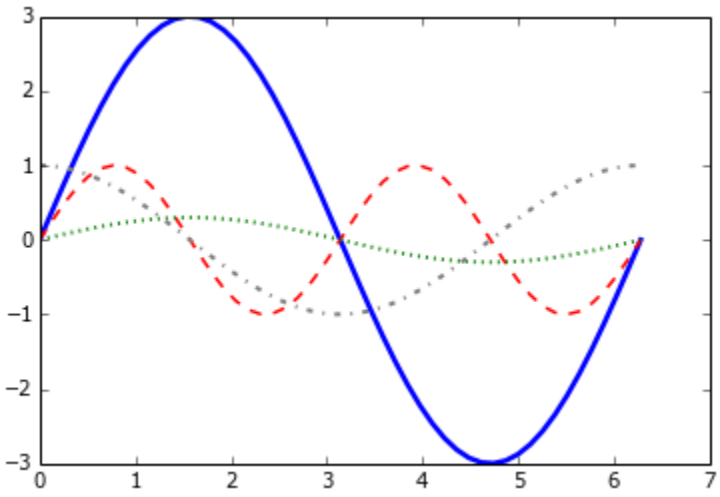
```
import matplotlib.pyplot as plt
import numpy as np
X = np.linspace(-2 * np.pi, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)
startx, endx = -2 * np.pi - 0.1, 2*np.pi + 0.1
starty, endy = -3.1, 3.1
plt.axis([startx, endx, starty, endy])
plt.plot(X,F1)
plt.plot(X,F2)
plt.plot(X,F3)
plt.plot(X, F1, 'ro')
plt.plot(X, F2, 'bx')
plt.show()
```



Changing the Line Style

The linestyle of a plot can be influenced with the linestyle or ls parameter of the plot function. It can be set to one of the following values:
`'-', '--', '-.', ':', 'None', ' ', ''`
We can use linewidth to set the width of a line as the name implies.

```
import matplotlib.pyplot as plt
import numpy as np
X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)
F4 = np.cos(X)
plt.plot(X, F1, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, F2, color="red", linewidth=1.5, linestyle="--")
plt.plot(X, F3, color="green", linewidth=2, linestyle=":")
plt.plot(X, F4, color="grey", linewidth=2, linestyle="-.")
plt.show()
```

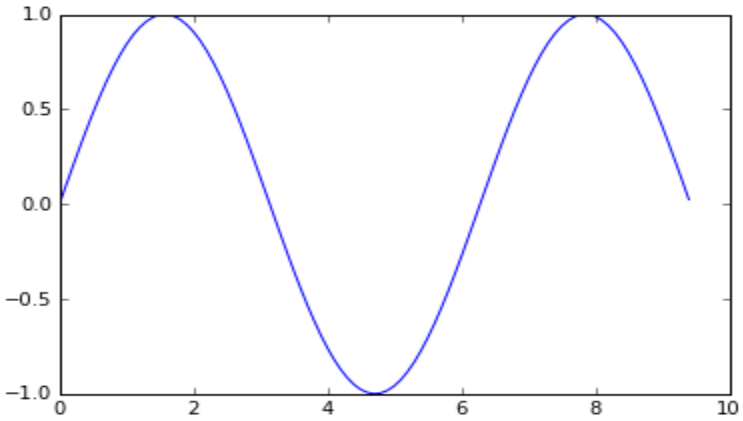


Another example:

```
import numpy as np
import matplotlib.pyplot as plt
```

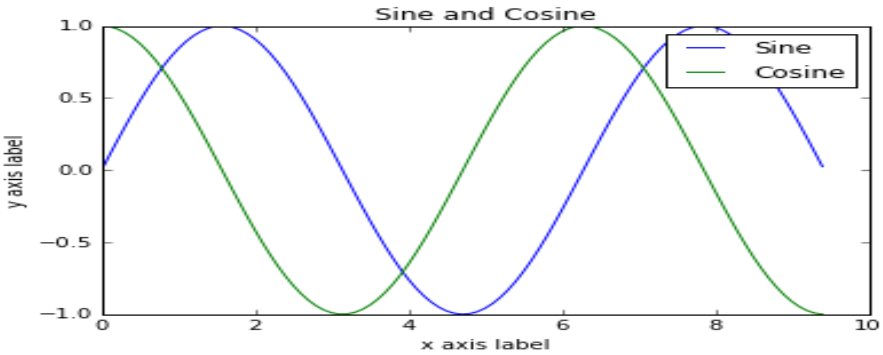
```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)
# Plot the points using matplotlib
plt.plot(x, y)
plt.show()  # You must call plt.show() to make graphics appear.
```

Running this code produces the following plot:



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

```
import numpy as np
import matplotlib.pyplot as plt
# 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)
# 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'])
plt.show()
```



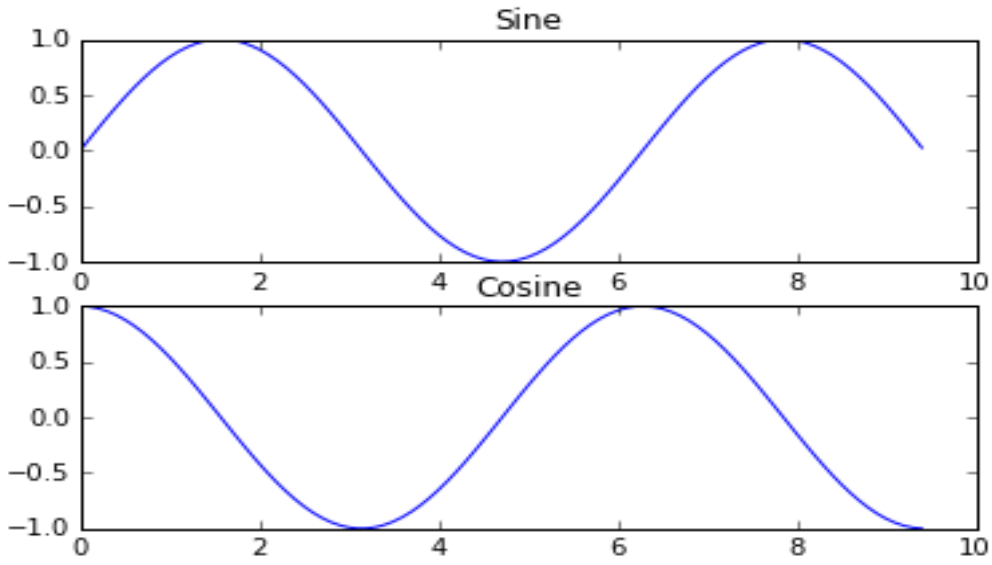
Subplots

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

```
import numpy as np
import matplotlib.pyplot as plt
# 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()
```

Running this code produces the following plot:



0.7 ToDo

This part will be given to you by the teacher assistant in the lab time.