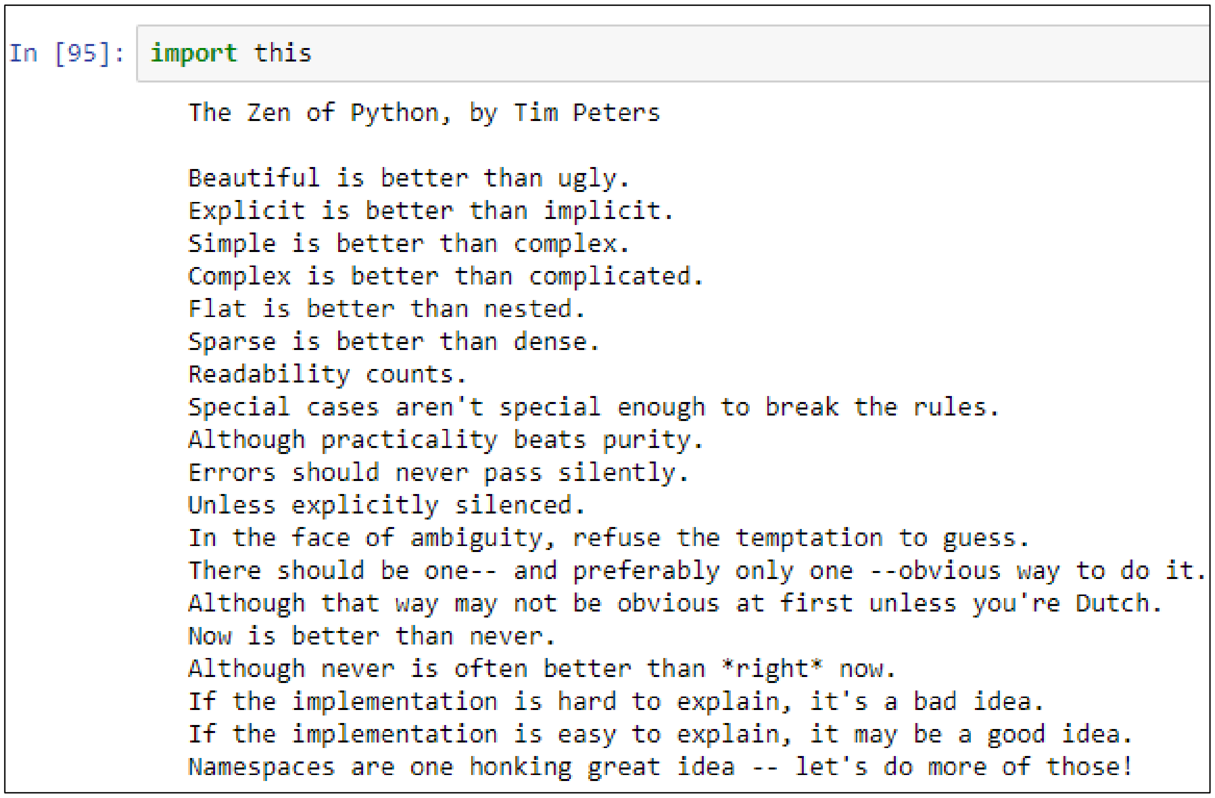
Applied Physics Lab -4

Importing

Python comes with lots of pre-made code baked in. These pieces of code are known as modules and packages. A module is a single importable Python file whereas a package is made up of two or more modules. A package can be imported the same way a module is. Whenever you save a Python script of your own, you have created a module. It may not be a very useful module, but that's what it is.

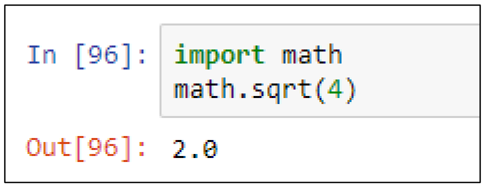
Import this

Python provides the **import** keyword for importing modules.

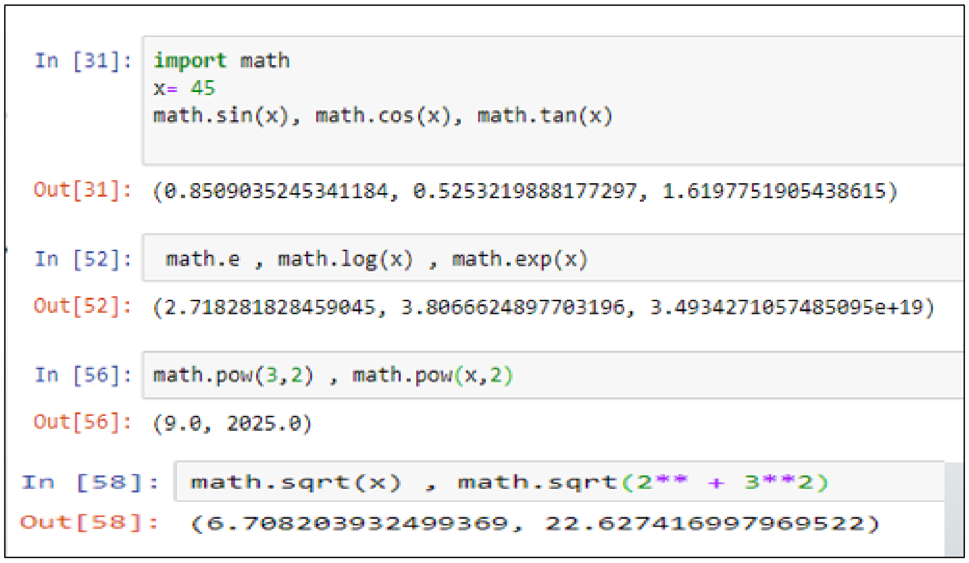


```
In [95]: import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

This module doesn't actually do anything, but it provided a fun little way to show how to import something. Let's actually import something we can use, like the math module:



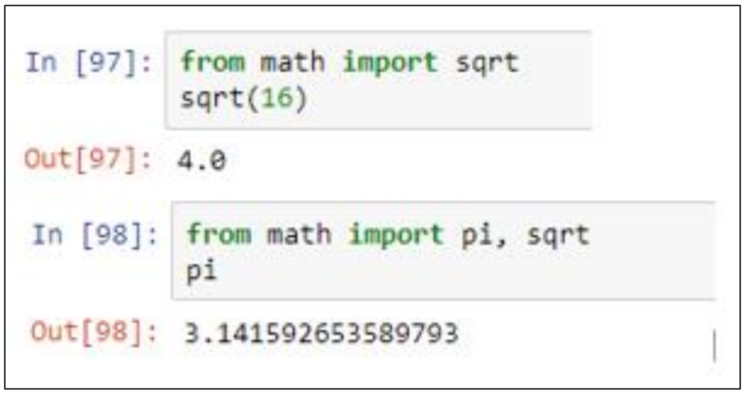
```
In [96]: import math
         math.sqrt(4)

Out[96]: 2.0
```



```
In [31]: import math
         x= 45
         math.sin(x), math.cos(x), math.tan(x)

Out[31]: (0.8509035245341184, 0.5253219888177297, 1.6197751905438615)

In [52]:  math.e , math.log(x) , math.exp(x)

Out[52]: (2.718281828459045, 3.8066624897703196, 3.4934271057485095e+19)

In [56]: math.pow(3,2) , math.pow(x,2)

Out[56]: (9.0, 2025.0)

In [58]: math.sqrt(x) , math.sqrt(2** + 3**2)
Out[58]:  (6.708203932499369, 22.627416997969522)
```

Using from to import

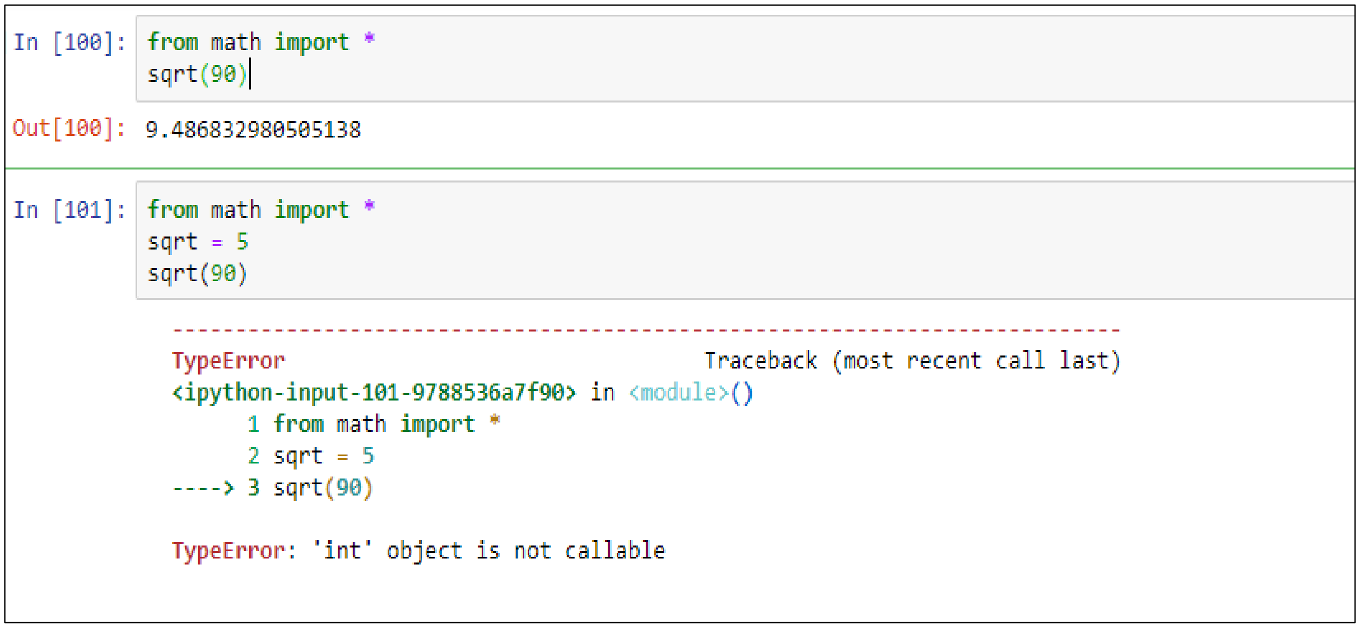
Some people don't like having to preface everything they type with the module name. Python has a solution for that! You can actually import just the functions you want from a module. Let's pretend that we want to just import the **sqrt** function:



```
In [97]: from math import sqrt
         sqrt(16)

Out[97]: 4.0

In [98]: from math import pi, sqrt
         pi

Out[98]: 3.141592653589793
```

Import everything

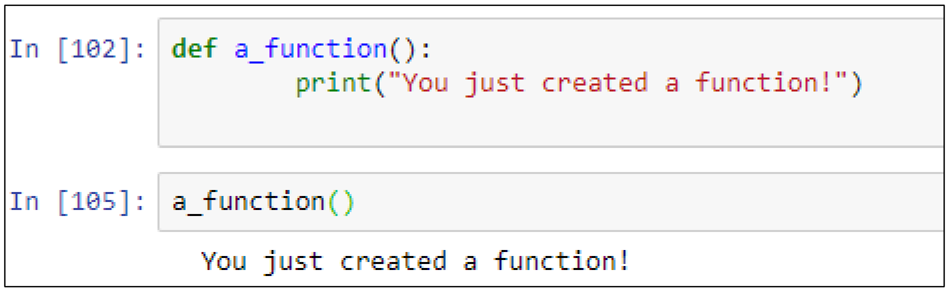
Python provides a way to import **all** the functions and values from a module as well. This is actually a **bad** idea as it can contaminate your **namespace**. A namespace is where all your variables live during the life of the program.



```
In [100]: from math import *
          sqrt(90)

Out[100]: 9.486832980505138

In [101]: from math import *
          sqrt = 5
          sqrt(90)

    ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)
    <ipython-input-101-9788536a7f90> in <module>()
          1 from math import *
          2 sqrt = 5
    ----> 3 sqrt(90)

    TypeError: 'int' object is not callable
```

Functions

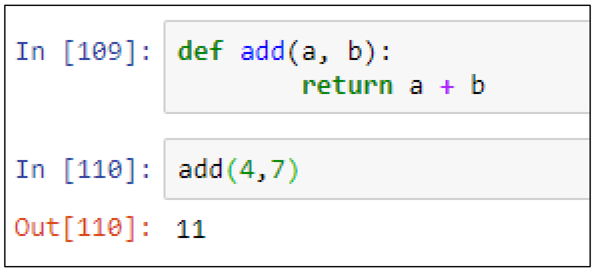
A function is a structure that you define. You get to decide if they have arguments or not. You can add keyword arguments and default arguments too. A function is a block of code that starts with the def keyword, a name for the function and a colon. Here's a simple example:



```
In [102]: def a_function():
                  print("You just created a function!")

In [105]: a_function()

            You just created a function!
```

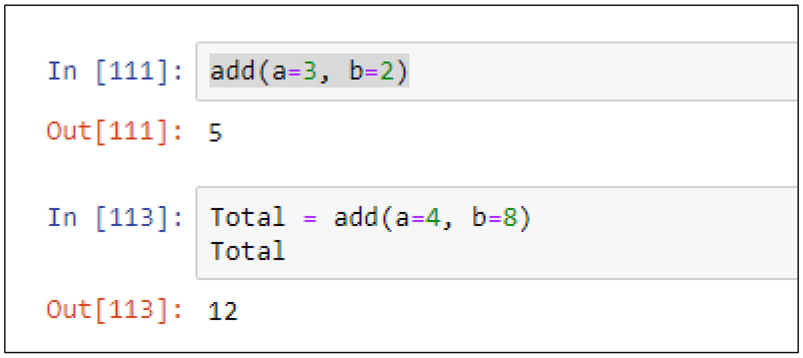
Passing Arguments to a Function

Now we're ready to learn about how to create a function that can accept arguments and also learn how to pass said arguments to the function. Let's create a simple function that can add two numbers together:

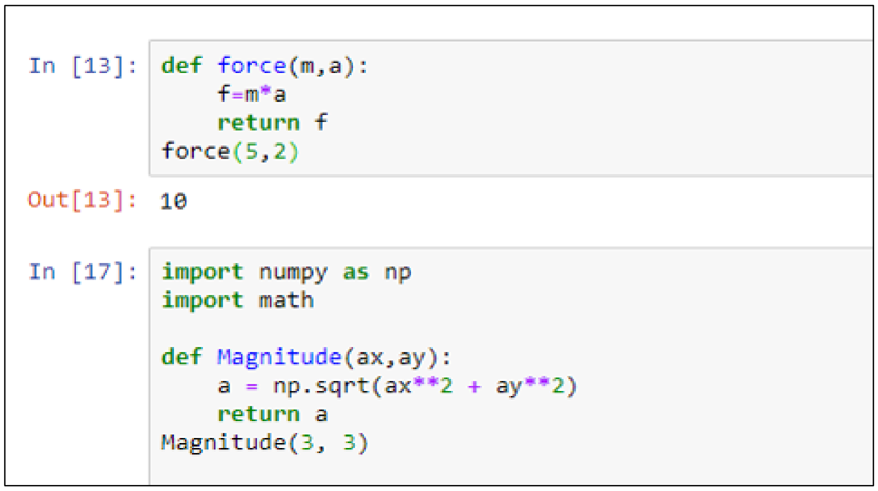


```
In [109]: def add(a, b):
                  return a + b

In [110]: add(4,7)

Out[110]: 11
```

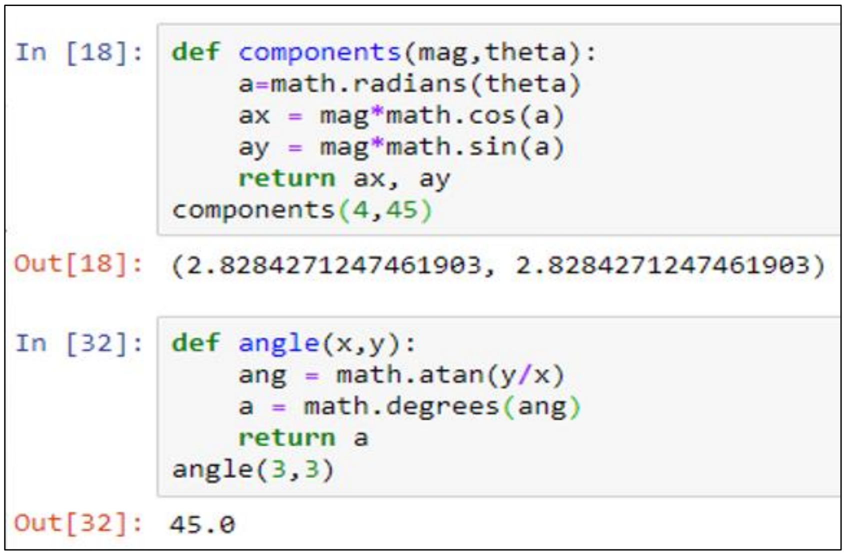
You can also call the function by passing the name of the arguments:



```
In [111]: add(a=3, b=2)

Out[111]: 5

In [113]: Total = add(a=4, b=8)
          Total

Out[113]: 12
```



```
In [13]: def force(m,a):
             f=m*a
             return f
         force(5,2)

Out[13]: 10

In [17]: import numpy as np
         import math

         def Magnitude(ax,ay):
             a = np.sqrt(ax**2 + ay**2)
             return a
         Magnitude(3, 3)
```

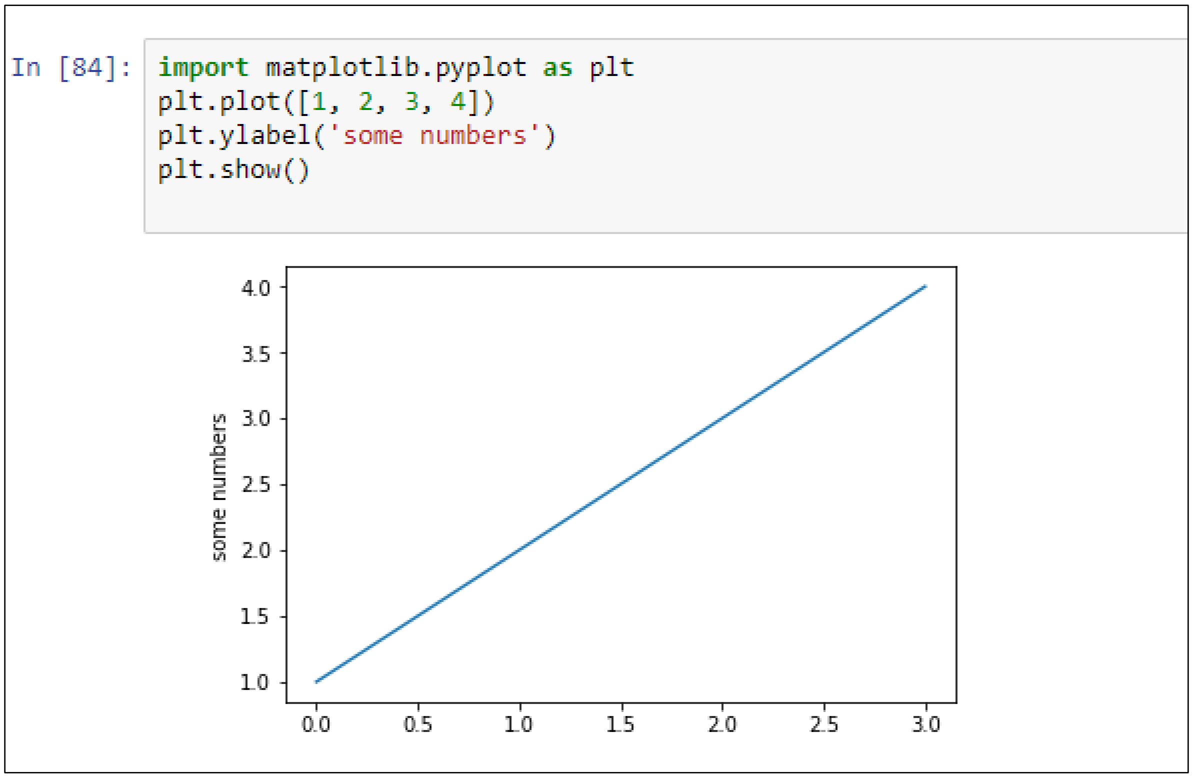


```
In [18]: def components(mag,theta):
             a=math.radians(theta)
             ax = mag*math.cos(a)
             ay = mag*math.sin(a)
             return ax, ay
         components(4,45)

Out[18]: (2.8284271247461903, 2.8284271247461903)

In [32]: def angle(x,y):
             ang = math.atan(y/x)
             a = math.degrees(ang)
             return a
         angle(3,3)

Out[32]: 45.0
```

Plotting in Python using Matplotlib

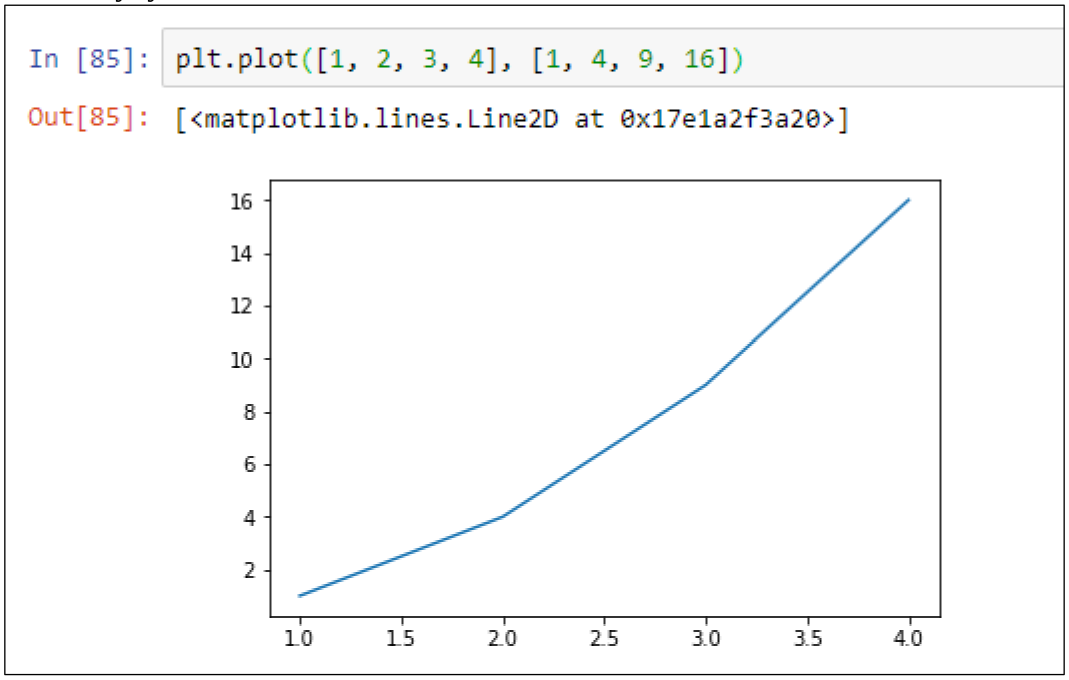
Matplotlib is an excellent 2D and 3D graphics library for generating scientific figures based on NumPy. Some of the many advantages of this library include:

- Easy to get started
- Support for formatted labels and texts
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS, and PGF.
- GUI for interactively exploring figures and support for headless generation of figure files (useful for batch jobs).



```
In [84]: import matplotlib.pyplot as plt
         plt.plot([1, 2, 3, 4])
         plt.ylabel('some numbers')
         plt.show()
```

Plot is a versatile function, and will take an arbitrary number of arguments. For example, to plot x versus y, you can write:

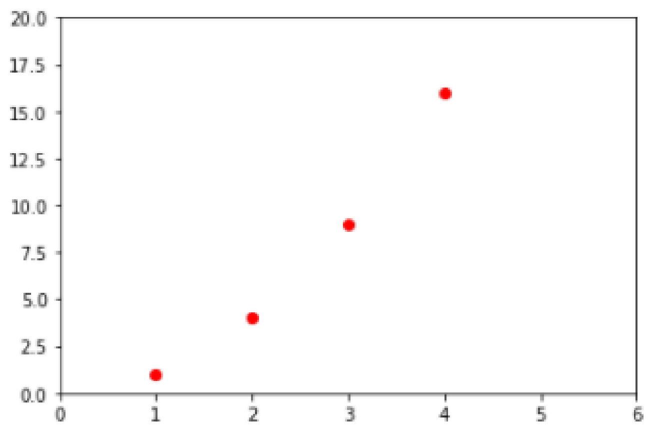


```
In [85]: plt.plot([1, 2, 3, 4], [1, 4, 9, 16])

Out[85]: [<matplotlib.lines.Line2D at 0x17e1a2f3a20>]
```

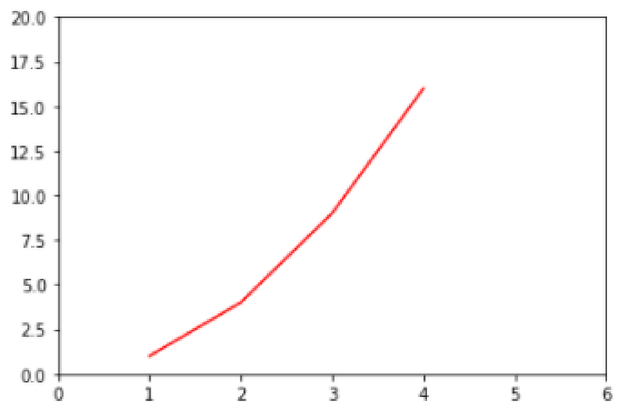
Formatting the style of your plot

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above with red circles, you would issue.

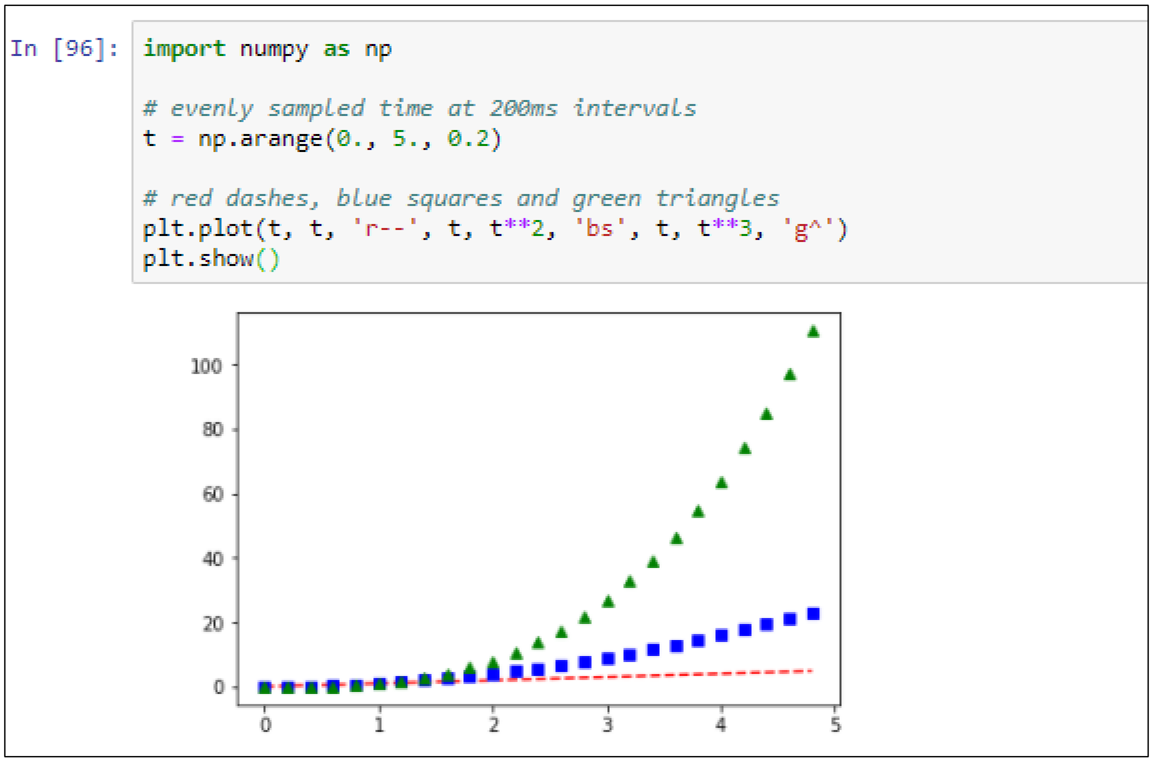
```
In [88]: plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')      # ro is for red circle pattern
         plt.axis([0, 6, 0, 20])
         plt.show()
```



```
In [90]: plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'r-')      # r- is for red solidline pattern
         plt.axis([0, 6, 0, 20])
         plt.show()
```



If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates plotting several lines with different format styles in one function call using arrays.

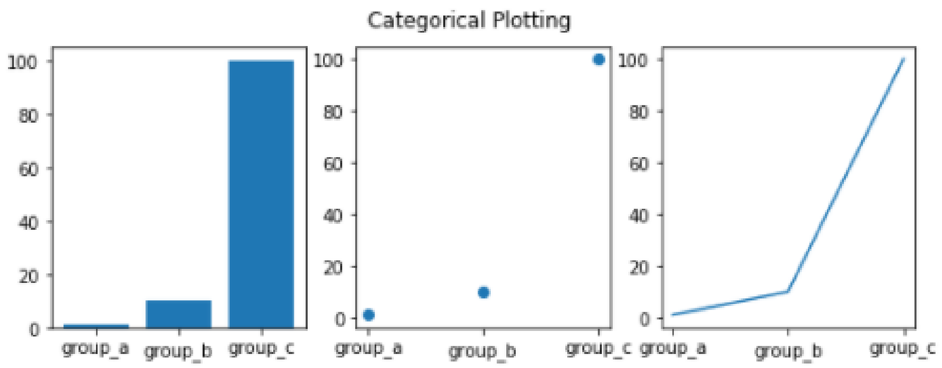


```
In [96]: import numpy as np

         # evenly sampled time at 200ms intervals
         t = np.arange(0., 5., 0.2)

         # red dashes, blue squares and green triangles
         plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
         plt.show()
```

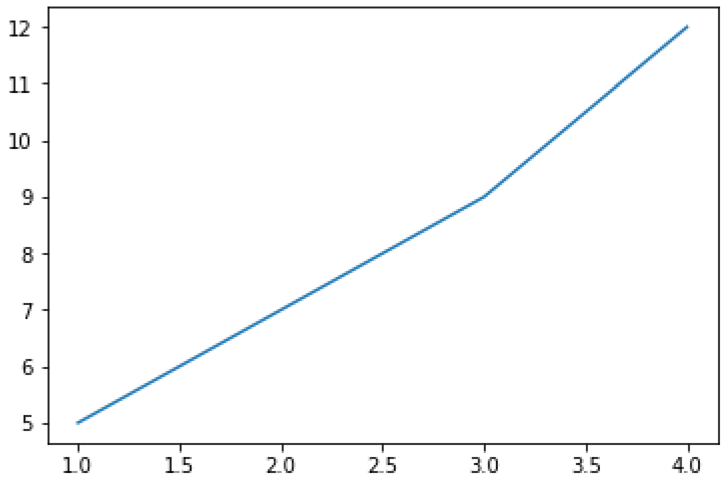
Plotting with categorical variables

It is also possible to create a plot using categorical variables. Matplotlib allows you to pass categorical variables directly to many plotting functions. For example:

```
In [97]: names = ['group_a', 'group_b', 'group_c']
         values = [1, 10, 100]

         plt.figure(figsize=(9, 3))

         plt.subplot(131)
         plt.bar(names, values)
         plt.subplot(132)
         plt.scatter(names, values)
         plt.subplot(133)
         plt.plot(names, values)
         plt.suptitle('Categorical Plotting')
         plt.show()
```

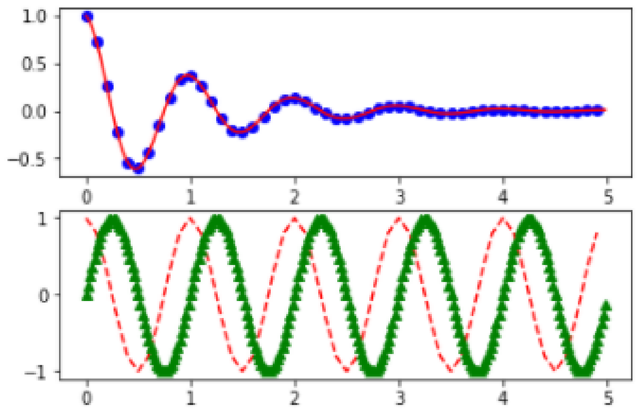


```
In [706]: import matplotlib.pyplot as plt

          x = np.array([1,2,3,4])
          y = np.array([5,7,9,12])

          plt.plot(x,y)
          plt.show()
```



Working with multiple figures and axes

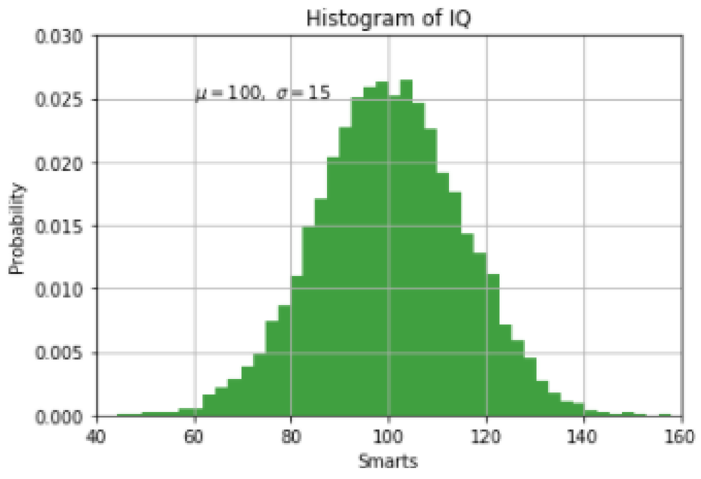
```
In [129]: def f(t):
              return np.exp(-t) * np.cos(2*np.pi*t)

          t1 = np.arange(0.0, 5.0, 0.1)
          t2 = np.arange(0.0, 5.0, 0.02)

          plt.figure()
          plt.subplot(211)
          plt.plot(t1, f(t1), 'bo', t2, f(t2), 'r')

          plt.subplot(212)
          plt.plot(t1, np.cos(2*np.pi*t1), 'r--', t2, np.sin(2*np.pi*t2), 'g^')
          plt.show()
```



Working with text

Text can be used to add text in an arbitrary location, and xlabel, ylabel and title are used to add text in the indicated locations.

```
In [131]: mu, sigma = 100, 15
          x = mu + sigma * np.random.randn(10000)

          # the histogram of the data
          n, bins, patches = plt.hist(x, 50, density=1, facecolor='g', alpha=0.75)


          plt.xlabel('Smarts')
          plt.ylabel('Probability')
          plt.title('Histogram of IQ')
          plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
          plt.axis([40, 160, 0, 0.03])
          plt.grid(True)
          plt.show()
```


Histogram of IQ
$\mu=100,\ \sigma=15$
Probability
Smarts