(DRAFT)
PYTHON FUNDAMENTALS II:
NUMPY & MATPLOTLIB

TROY P. KLING

Contents

1. Importing Libraries

In order to make use external libraries in Python, one uses the `import` statement. Since this tutorial focuses on `numpy` and `matplotlib`, the first thing to do is import these libraries using the following two lines of code.

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

Note that Python allows you to rename libraries when you import them – from here on out, the Python code in this tutorial will reference `np` rather than `numpy` and `plt` rather than `matplotlib.pyplot`. This is just shorthand notation to make it easier to write code – you can name the libraries anything you want, or simply use the default names.

There are numerous other useful libraries in Python. Most of the libraries that are commonly used for scientific purposes like mathematical programming, machine learning, and image processing (e.g. `numpy` and `matplotlib`) are included in a special Python distribution called Anaconda. For this reason and others, the author strongly encourages the usage of Anaconda over other Python distributions.

Python also contains several "easter egg" libraries. For example, try running the following code, one line at a time.

```
import this
import __hello__
from __future__ import braces
import antigravity
```

2. Introduction to numpy

`numpy` is a Python library designed to efficiently handle large, multi-dimensional arrays. It is comparable to MATLAB, and much of the syntax and function naming schemes in `numpy` were built with this similarity in mind.

The most important data structure in `numpy` is the `ndarray` (i.e. n-dimensional array). There are several ways to initialize an `ndarray`. One can start with a list and cast it as an `ndarray`.

```
list1 = [[0,1,2,3,4], [5,6,7,8,9], [10,11,12,13,14]]
arr1 = np.array(list)
```

One can also specify a range of values and then reshape the single-dimensional array into a two (or more) dimensional array.

```
arr2 = np.arange(15).reshape(3,5)
```

Or one could start with an empty array, an array of zeros/ones, or an array of random elements.

```
arr3 = np.empty((3,5))
arr4 = np.zeros((3,5))
arr5 = np.ones((3,5))
arr6 = np.random.rand(3,5)
```

The most commonly used function for initializing an array is `np.zeros()`, but the other functions have their uses as well. However one initializes an array, the important thing to keep in mind is that arrays in `numpy` have a *predetermined, fixed size.* Lists, on the other hand, do not. This means that once an array is initialized, it's somewhat difficult to modify its size.

Another useful method for initializing an array is the `np.linspace()` method, which produces an array of evenly (linearly) space values. For example, if I want an array of the form `[3, 3.5, 4, 4.5, 5, 5.5, 6]`, I could execute the following command.

```
arr7 = np.linspace(3, 6, 7)
```

The first argument is the starting number, the second argument is the ending number, and the third argument is the number of elements desired. This gives us all the tools we will need for building arrays in `numpy`. Next, we must learn some common array operations. Start with a couple one-dimensional arrays.

```
a = np.array([20,30,40,50])
b = np.arange(4)
```

Arithmetic operations applied to arrays in `numpy` behave *element-wise*. For adding or subtracting two arrays, this is usually the desired behavior, but for multiplying arrays it may not be. Make sure to keep this in mind when performing array arithmetic.

```
sum = a + b
diff = a - b
prod = a * b
quot = a / b
power = a ** b
```

Using these built-in arithmetic operations tends to be faster than writing loops to do the same thing, because most operations in `numpy` are written in CPython and vectorized.

One can perform other basic mathematical/statistical operations as well. Below are just a few of the available functions.

```
sum = a.sum()          # Or np.sum(a)
mean = a.mean()        # Or np.mean(a)
std_dev = a.std()      # Or np.std(a)
max = a.max()          # Or np.max(a)
min = a.min()          # Or np.min(a)
```

It is important to note that if one is working with multi-dimensional arrays, the functions above may not produce the desired output. For example, one might wish to find the sum of each row of a two-dimensional array, but calling `np.sum()` will compute the sum of *all* elements in the array. To apply functions to rows, columns (or higher-dimensional cuts) of an array, the `axis` argument is used.

```
c = np.arange(20).reshape((4,5))
col_sum = c.sum(axis=0)             # Or np.sum(c, axis=0)
row_sum = c.sum(axis=1)             # Or np.sum(c, axis=1)
```

`numpy` has implemented numerous other mathematical functions – far too many to list here. A sample of its mathematical arsenal is given below.

```
np.abs(c)       # Absolute value
np.floor(c)     # Floor function
np.sqrt(c)      # Square root
np.exp(c)       # Exponentiation
np.log2(c)      # Logarithm base-2
np.sin(c)       # Sine
np.tanh(c)      # Hyperbolic tangent
np.i0(c)        # Bessel function of the first kind, order 0
np.trapz(c)     # Integration using trapezoid rule
```

Indexing and shape manipulation are also important concepts to master. Indexing arrays works very similar to how it does for lists.

```
arr_sin = np.sin(np.linspace(0, 2*np.pi, 9)).reshape((3,3))
arr_cos = np.cos(np.linspace(0, 2*np.pi, 9)).reshape((3,3))
arr_sin[1, 2]
arr_sin[0:2, 1]
arr_sin[:, 1]
arr_sin[0:2, :]
```

We already know that `numpy` arrays can be reshaped. They can also be flattened (turned into one-dimensional arrays), transposed, and stacked horizontally and vertically.

```
arr_sin.shape
arr_sin.flatten()                 # Make one-dimensional
arr_sin.T                         # Or np.transpose(arr_sin)
np.hstack((arr_sin, arr_cos))     # Horizontal stack
np.vstack((arr_sin, arr_cos))     # Vertical stack
```

This concludes the section on `numpy`. The skills taught in this section will be critically important for the next one, since `matplotlib` depends so heavily on `numpy`.

3. Introduction to matplotlib

matplotlib is another Python library intended to mimic certain elements of MATLAB's functionality. The pyplot package within matplotlib is particularly useful for producing MATLAB-like plots. It can be imported used the following command.

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

The most important commands in matplotlib are plt.plot() and plt.show(). The former command creates a line or scatter plot and the latter one displays it on the screen. Of course, plt.plot() requires several arguments, since it needs to know exactly what is being plotted. Consider the following four plots.

```
plt.plot([1,2,3,4])
plt.ylabel("Some numbers")
plt.show()

plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.xlabel(‘‘x values’’)
plt.show()
```

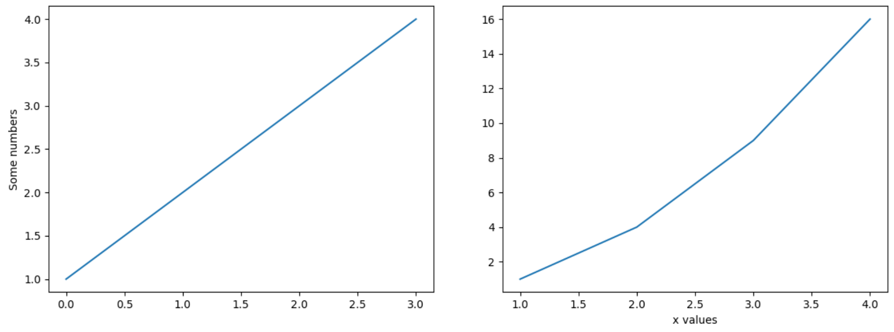

Figure 1. Line plots using four data points.

```
plt.plot([1,2,3,4], [1,4,9,16], ’ro’)
plt.title(‘‘Scatter plot’’)
plt.axis([0, 6, 0, 20])
plt.show()

t = np.arange(0., 5., 0.2)
plt.plot(t, t, ’r--’, t, t**2, ’bs’, t, t**3, ’g^’)
plt.show()
```

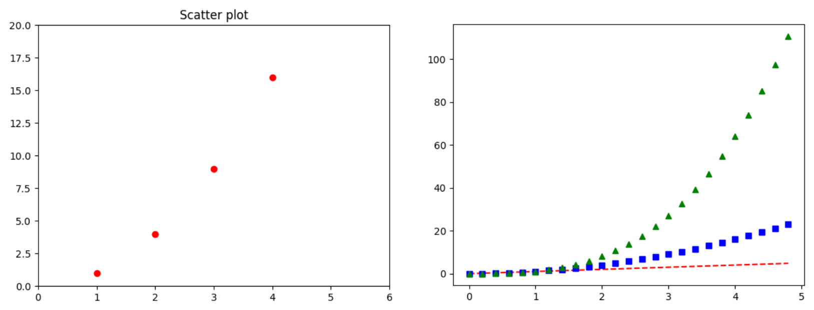

FIGURE 2. Scatter plots with several data points.

This code produces four plots, each of which opens in a separate window. This is often undesirable behavior when several plots are being made. Instead, we would like to group several plots together in a single window. The can be accomplished with the `subplot` command. We begin by defining two functions, $f(x)$ and $g(x)$, which are unimportant in the grand scheme of things, and then create four plots.

```
def f(x):
    return np.exp(-x) * np.cos(2*np.pi*x)

def g(x):
    return np.sin(2*np.pi*x) / x

x1 = np.arange(0.0, 2*np.pi, 0.25)
x2 = np.arange(0.0, 2*np.pi, 0.02)

plt.subplot(221)
plt.plot(x1, f(x1), 'bo', x2, f(x2), 'k')

plt.subplot(222)
plt.plot(x2, np.cos(x2), 'r--')

plt.subplot(223)
plt.plot(x1, g(x1), 'mo', t2, g(x2), 'k')

plt.subplot(224)
plt.plot(x2, np.sin(x2), 'g--')

plt.show()
```

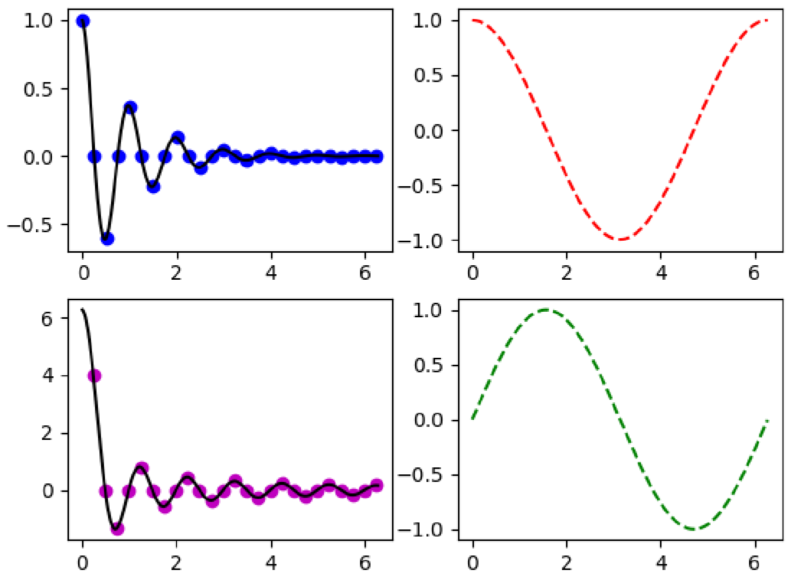

FIGURE 3. Four different functions, all plotted in the same window.

`matplotlib` has many different plotting commands available. We will see one more example – a histogram – which can be produced using the `plt.hist()` command. We will also make use of `numpy`'s `random` package to simulate a normal distribution of IQ scores.

```
mu = 100
sigma = 15
x = mu + sigma*np.random.randn(10000)
plt.hist(x, 50, normed=1, facecolor="g", alpha=0.75)

plt.title("Histogram of IQ")
plt.xlabel("Smarts")
plt.ylabel("Probability")
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

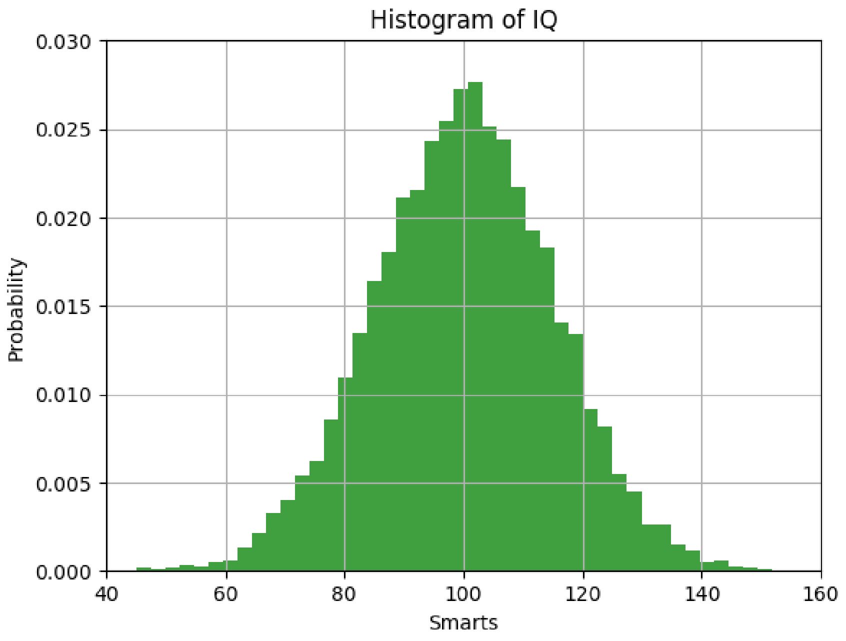

Figure 4. A histogram of simulated IQ scores.

4. Image Processing

Image processing is where you process images and stuff.

5. The Mandelbrot Set

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

def limit(c, n=25, p=2):
    z = 0
    for i in range(1, n):
        z = z**p + c
        if np.abs(z) > 2:
            return np.inf

    return np.abs(z)

def mandelbrot(n=25, size=250, xlim=(-2,2), ylim=(-2,2), p=2):
    x = np.linspace(xlim[0], xlim[1], size)
    y = np.linspace(ylim[0], ylim[1], size)
    m = np.zeros((size, size))

    for i in range(size):
        for j in range(size):
            z = limit(x[i] + y[j]*1j, n, p)
            if z < 2:
                m[j,i] = 1

    return m

def mandelbrot_color(n=25, size=250, xlim=(-2,2), ylim=(-2,2), p=2):
    x = np.linspace(xlim[0], xlim[1], size)
    y = np.linspace(ylim[0], ylim[1], size)
    m = np.zeros((size, size))

    for i in range(size):
        for j in range(size):
            c = x[i] + y[j]*1j
            z = 0
            for k in range(n):
                z = z**p + c
                if np.abs(z) < 2:
                m[j,i] += 1
                else:
                    break

    return m
```

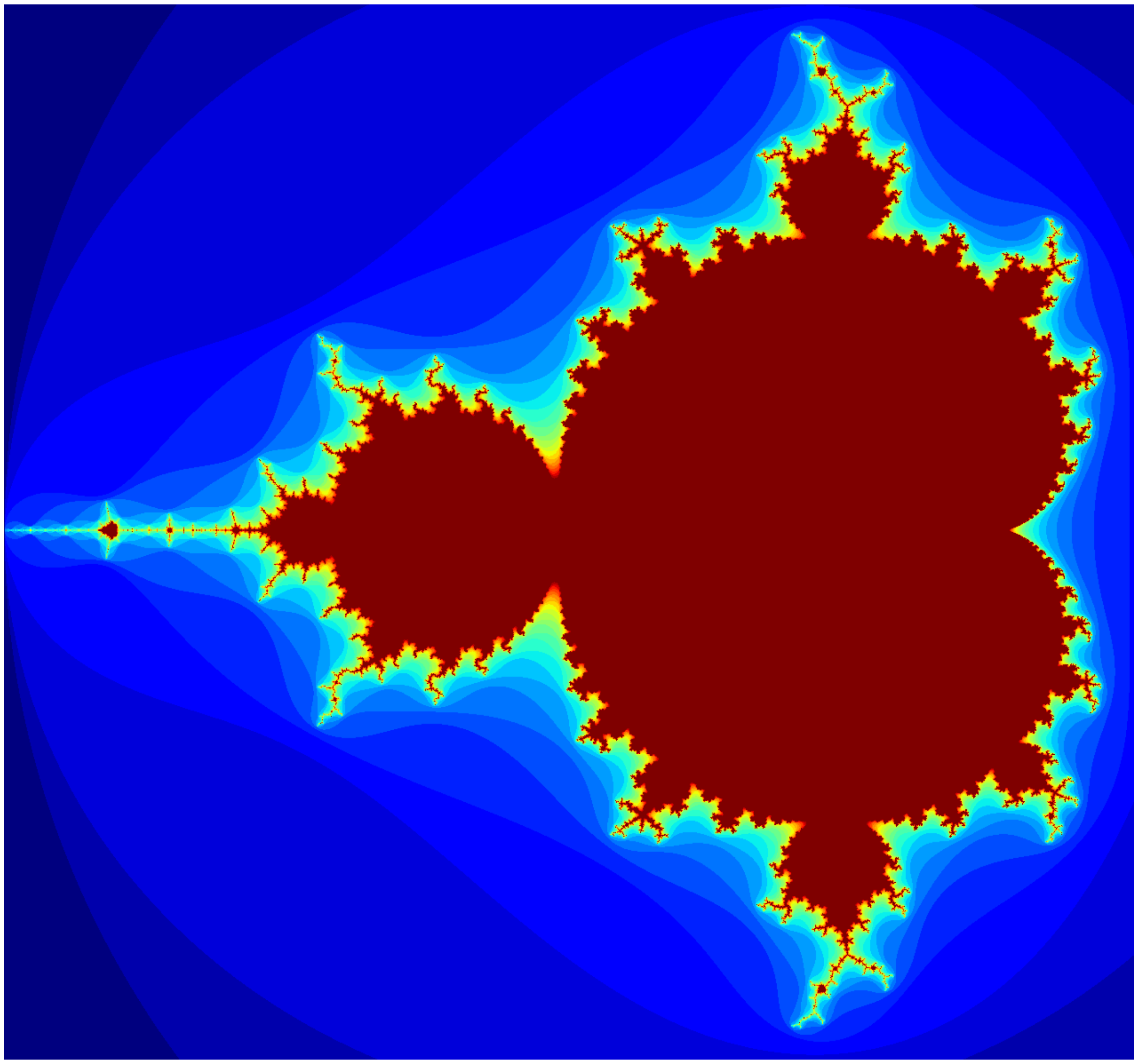

FIGURE 5. The resulting Mandelbrot set.

6. CONCLUSION

In conclusion, Python is awesome.