Chapter 9: Graphics and Interaction

This chapter will introduce concepts of graphics as well as objects in Python. Chapter 10 will cover objects and classes in detail.

9.1 Setting up the environment

As with all programming, the first step is to set up the environment needed to meet the project goals. In this case, the goal is to be able to create graphics and to use appropriate graphing and numerical packages. There are many graphing and data visualization packages and libraries available for Python. To get started, and in this chapter, the following two packages will be imported and utilized: **NumPy** and **MatPlotLib**.

NumPy resources and references can be found here: https://www.scipy.org/.

A direct reference and coding resource for **MatPlotLib** is located here: http://MatPlotLib.org/index.html. The package, MatPlotLib was created by John Hunter (1968-2012). It is requested on the MatPlotLib site, that if MatPlotLib contributes to a project that leads to a scientific publication, that proper acknowledgement be included.

To set up the environment for creating graphics in Python, the first step is to update some settings inside of Spyder.

Example 9.1: Updating Settings in Spyder

Follow these steps to update the required settings in Spyder to view graphics.

1) Inside of Spyder, choose Tools, and then choose Preferences.
2) From there, on the left, choose the IPython Console.
3) Next, at the top, choose the tab called Graphics.
4) From there, assure that *Activate Support* and *Automatically load Pylab and NumPy modules* are both checked.
5) Under the next area called, "Graphics backend", choose *Automatic* (unless you have a preference for Tkinter which is not discussed in this text).

Figure 10.1 will illustrate these above steps.

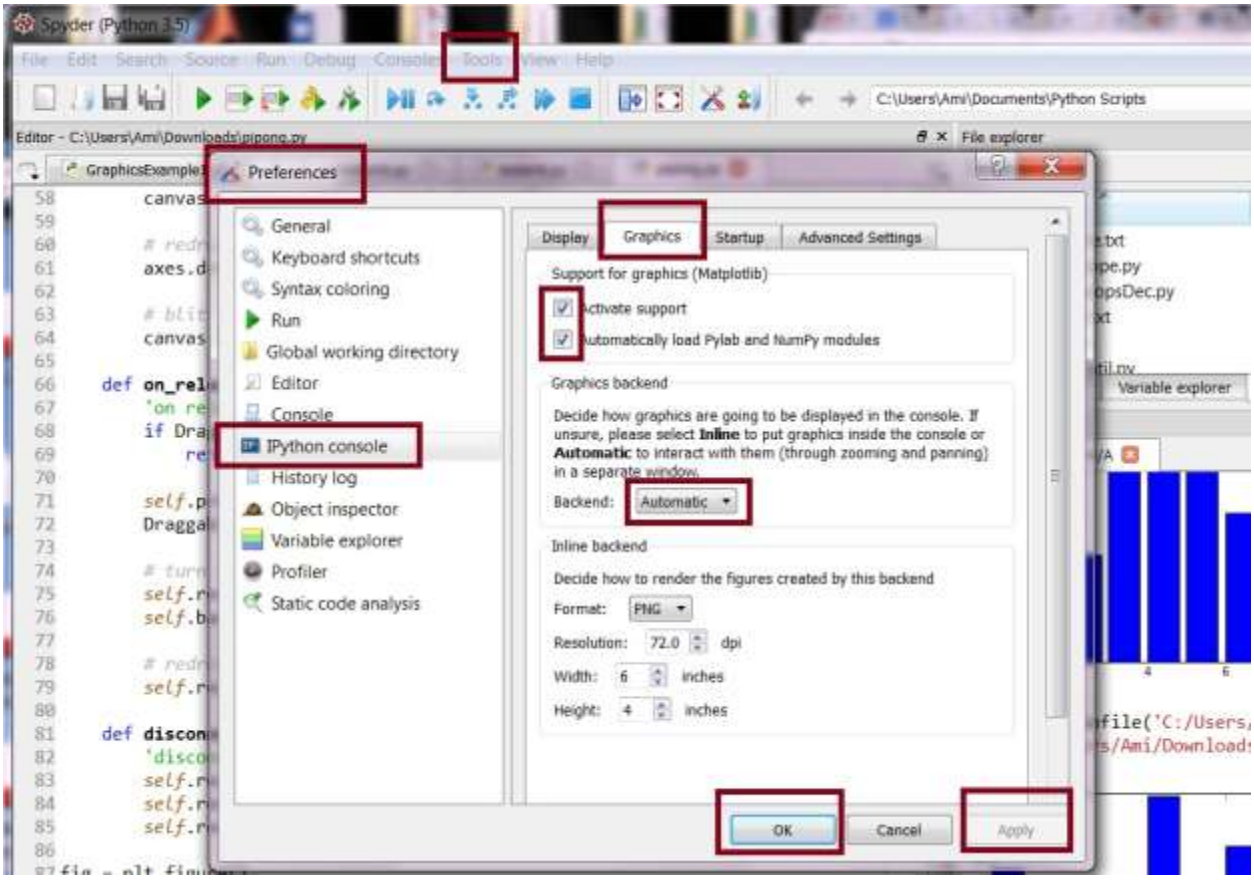


Figure 10.1: Setting up Spyder for Graphics

Once Spyder is updated for graphics, the next step is to test the environment with a small program that will use NumPy and MatPlotLib.

Exercise 10.1: Confirming the graphing environment in Spyder/Python

To complete this confirmation, follow these steps.

1) Create a new file in Spyder and save it as *GraphicsTest1.py*.
2) Type into that file the following program shown below.
3) Then save it and run it.
4) The program and its output are displayed below.

```
# GraphicsTest1.py
# Chapter 9
# Ami Gates

#Import numpy and give it the smaller name of np
#Import the method of matplotlib called pyplot and call it
mpp
import numpy as np
import matplotlib.pyplot as mpp
```

```
#Create a function called Quad that creates quadratic data.
# The function Quad takes two parameters. The first
parameter is
# an array of data called x. The second and optional
parameter is c
def Quad(x,c=0):
    y=x**2 + c
    return y

# Use NumPy arrays and the arrange method to create arrays
of numbers.
# Here, x1 is an array or set of numbers from -10 to 10 in
steps of .5
# x2 is a an array of numbers from -5 to 5 in steps of .3
x1 = np.arange(-10.0, 10.0, .5)
x2 = np.arange(-5.0, 5.0, .3)


#Create a plot called Figure 1
mpp.figure(1)

#In the plot called Figure 1, create a subplot with 2 rows,
2 columns,
# and place the subplot in location 1
mpp.subplot(221)
#On that subplot, plot the following. bo is blue o's
mpp.plot(x1, Quad(x1,-10), 'bo')
mpp.title("Parabola1")

#Create anther subplot and put it in location 2
# g^  are green triangles
mpp.subplot(222)
mpp.plot(x2, Quad(x2,4), 'g^')
mpp.title("Parabola2")
#Create anther subplot and put it in location 3
# r-- is red dashed line
mpp.subplot(223)
mpp.plot(x2, Quad(x2,10), 'r--')
mpp.title("Parabola3")

#Show the plot
mpp.show()
```

The output from running the above program should create a **new window** that will look like this:

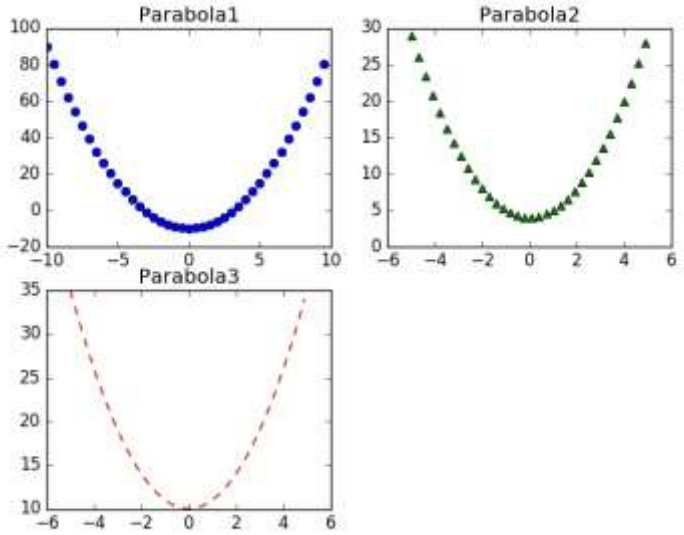


Figure 9.1: The output from GraphicsTest1.py above

If needed, close and re open Spyder and follow the steps until the program generates the desired result.

9.2: Line Graphs in Python

9.2.1: Importing packages: NumPy and MatPlotLib

Two Python packages will be used to create graphics. The first is **NumPy** and the second is **MatPlotLib**. Specifically, the **pyplot** module in MatPlotLib will be accessed. The **import** statement is used to bring packages, modules, and libraries into a Python program.

The syntax for import is:

```
import <module/package name> as variable
```

For example

```
import numpy as np
```

The `as` operator is optional. Therefore, the statement,

```
import numpy
```

will accomplish the same goal. The reason for using the `as` operator is to give NumPy a shorter "nickname" that can then be used throughout the program. To import the pyplot module from MatPlotLib, the statement is:

```
import matplotlib.pyplot as mpp
```

Again the "`as mpp`" portion is optional, and the name, `mpp`, can be any legal Python name.

9.2.2: Line Graphs and MatPlotLib methods

Python allows for many plotting and graphing options. Recall that a standard two-dimensional plot is a collection of (x,y) pairs that have each been placed on a Cartesian coordinate system.

The **line graph** will plot any finite or infinite collection of (x,y) points.

The syntax for the line graph plot is:

```
matplotlib.pyplot.plot(x values, y values, linetype)
```

Because the variable name, `mpp` is being used to represent `matplotlib.pyplot`, the above statement can be written as:

```
mpp.plot(x values, y values, linetype)
```

There are several line types. A few are "ro", for "red dots", "g^" for "green triangles", and "y-" for "yellow solid line".

The plot method can accept x values and y values as lists, ranges, or functions. As will be illustrated shortly, using NumPy allows for the use of NumPy arrays and mathematical functions.

To create a line graph, several methods will be used. Some of these methods include **plot**, **axis**, **title**, and **show**.

The plot method:

The syntax for the plot method is above and is:

```
matplotlib.pyplot.plot(x values, y values, linetype)
```

And by using mpp to represent matplotlib.pyplot, this can be written as:

```
mpp.plot(x values, y values, linetype)
```

There must be the same number of x values as there are y values, just as there would be if there were a collection of (x,y) pairs. The x and y values can be finite lists, can be ranges, or can be functions of each other.

For example, these statements will create a plot of three points, (1,4), (2,5), and (3,6). The points will be red dots as designated by "ro".

```
xvals=[1, 2, 3]
yvals=[4, 5, 6]
linetype="ro"
mpp.plot(xval, yval, linetype)
```

This same plot can be created with this statement:

```
mpp.plot([1,2,3], [4,5,6], 'ro')
```

Instead, the x values might be a range from a to b and the y values might be a function of that set of numbers. For example:

```
xvals=arange(0,10,1)
yvals=xvals**2
```

In this case, the x values are [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] and the y values are [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

The **arange(a,b,c)** function will create a range or collection of numbers that start at "a", end at "b", and are separated by the value "c". If "c" is not included, the default is "1".

The axis method:

The axis method will create the x-y axis and will designate the range of each axis. The syntax is:

```
mpp.axis([a,b,r,s])
```

In this case, the x axis will range from a to b and the y axis will range from r to s.

The title method:

The title method places a title on the graph. The syntax is:

```
mpp.title(<string>)
```

The show method:

The show method will render or create the graph. The syntax is:

```
mpp.show()
```

The next example will illustrate the above methods and the creation of a line graph.

Example 9.1: Making a line graph in Python

The following small program will create a line graph. It is recommended that you type in, save, and run this program. Confirm your results with the output below.

```
import matplotlib.pyplot as mpp

mpp.plot([1,2,3], [4,5,6], 'ro')
mpp.axis([0,10,0,10])
mpp.title("Example Small Plot")

#Show the plot
mpp.show()
```

In the above example program, the first statement imports the module called `matplotlib.pyplot` and assigns it a "short name" of `mpp` (for ease of use in the program). The following statement,

```
mpp.plot([1,2,3], [4,5,6], 'ro'),
```

calls on the **plot** method. The first parameter of the call is `[1,2,3]`, which is a list of "x" values. The second parameter is `[4,5,6]` which is the corresponding list of "y" values. These two lists of values are equivalent to the (x,y) pairs: (1,4), (2,5), and (3,6). The third parameter, '`ro`' stands for "red dots" for the plot points..

The next statement, `mpp.axis([0,10,0,10])` determines the range of the x axis (which is defined here as starting at 0 and ending at 10) and the range of the y axis (also started at 0 and ending at 10). The statement, `mpp.title()` allows for naming of the plot. Finally, `mpp.show()` will create the graph. The result of this example program is illustrated in Figure 9.2

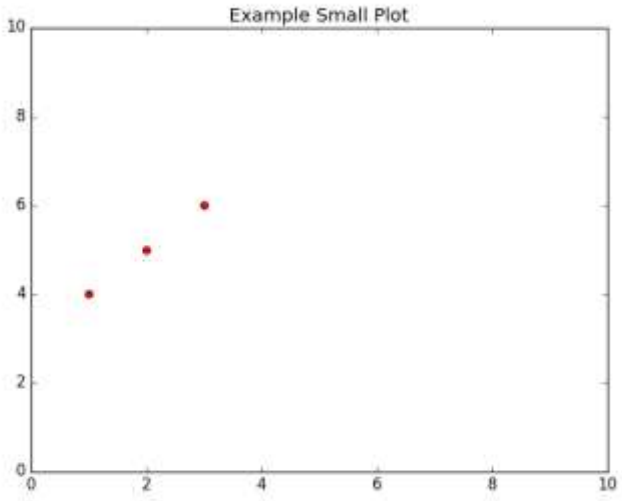


Figure 9.2: The output of Example 9.1

9.3: NumPy, Arrays, and Graphing

While MatPlotLib can create excellent graphics, the NumPy package will allow for the use of a data structure called a **NumPy array.** Rather than using finite lists, such as [1, 2, 3] for x values and [4,5,6] for y values, it is often necessary to create ranges of data and functions of those ranges.

9.3.1: NumPy and arrays

By importing and using NumPy, it is possible to create an array of values using **arange**.

The syntax for arange is:

```
ArrayOfEvenlySpacedValues=arange(start, stop, step, dtype)
```

The `start` parameter is a numerical value and is optional. If the start parameter is omitted, the default is 0.

The `stop` parameter is required. The range or interval of numbers will end **not including** `stop`. This can be noted mathematically as, [`start, stop`).

The `step` parameter is optional and is a numerical value and represents the space or distance between each number in the range. The default is 1.

The `dtype` parameter is optional and specifies the data type of the array being created. If the dtype is omitted, the "type" of the array will be inferred based on `start` and `stop`. For

example, if start and stop are both integer values, the type of the array returned by `arange` will be of integer type.
The following are several examples of using **arange** from **NumPy** to create arrays.

```
import numpy as np

MyIntArray=np.arange(0,10,1)

MyIntArray
Out[112]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

MyIntArray2=arange(10)

MyIntArray2
Out[125]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

MyIntArray3
Out[129]: array([-50, -40, -30, -20, -10,   0,  10,  20,
30,  40])

MyFloatArray=np.arange(-12.3,13.2, 4.5)

MyFloatArray
Out[122]: array([-12.3,  -7.8,  -3.3,   1.2,   5.7,  10.2])
```

One of the values of NumPy arrays, and what differentiates NumPy arrays from other data types such as lists or tuples, is that they can be manipulated via mathematical functions. In other words, if x values are created using a NumPy array, y values can be created directly as a function of x.

Example 9.3.1: Creating x and y values using numpy arrays

Many types of graphs require x and y values. Recall that the `plot` method offered by matplotlib.pyplot has the form, mpp.plot(x values, y values, linetype). In the above examples, the x values and corresponding y values were represented as finite lists. An alternative is to use a NumPy array to represent the x values and then to create a mathematical function of the array to create the corresponding y values. This example will illustrate that option and will compare both the list method and the array method.

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

# The list method to create x values x1 and y values y1
```

```
x1=[1,2,3,4,5]
y1=[2,3,4,5,6]
# The numpy array and function method to create
# x values x2 and y values y2

x2=np.arange(1,5,1)
y2=x2+1

mpp.figure(1)

## This is a subplot of the list (x,y) values
mpp.subplot(121)
mpp.plot(x1, y1, 'ro')
mpp.title("Plot Using Lists")
mpp.axis([0,7,0,7])

## This is a subplot of the array/function (x,y) values
mpp.subplot(122)
mpp.plot(x2, y2, 'bo')
mpp.title("Plot Using NumPy Arrays")
mpp.axis([0,7,0,7])

#Show the plot
mpp.show()
```

The output for the above program is:

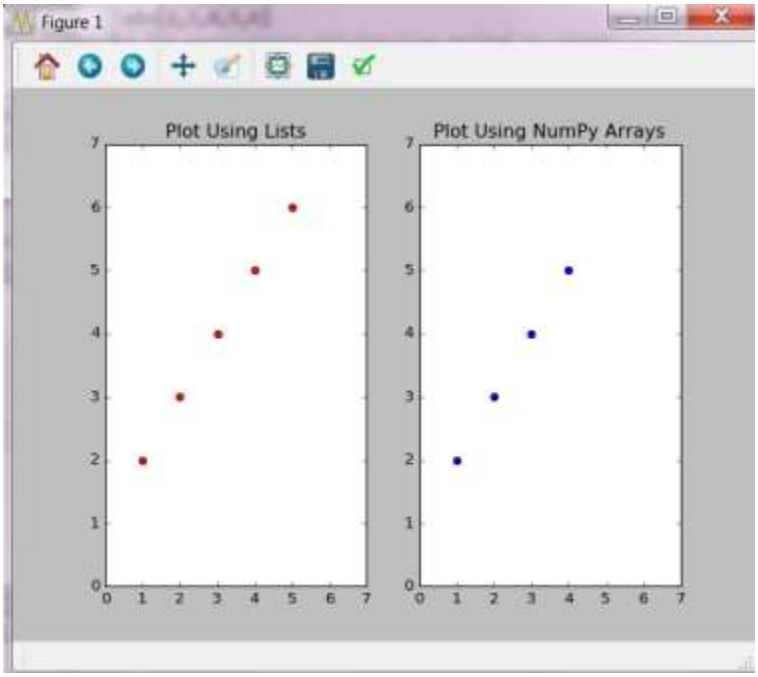


Figure 9.3: Output for example 9.3.1

In reviewing the above example, first consider these four statements from the program.

```
x1=[1,2,3,4,5]
y1=[2,3,4,5,6]

x2=np.arange(1,5,1)
y2=x2+1
```

Here, `x1` is a list of values from 1 through 5 and `y1` is the list of values from 2 through 6. These statements use the **list** structure to define the x and y values. Alternatively, x2 and y2 are defined using the NumPy method, **arange** and creates an array of values from 1 to 5, not including the 5, and with a step of 1 (which means that the gap between each value in the array is 1). Therefore, `x2` is `array([1, 2, 3, 4])`.

Next, `y2` can be defined as a function of `x2`, where `y2 = x2 + 1`. Therefore, `y2` is `array([2, 3, 4, 5])`. It is very important to note here that lists do not offer this same functionality. It would not be possible to have the statement, y1 = x1 + 1, because x1 is a list, not an array. For this reason, and many others, the use of NumPy arrays is common and valuable when employing Python graphics and visualization techniques.

The next statement in the program example above is, `mpp.figure(1)`. This statement will label the new figure as **Figure 1** (which can be seen in the above output for the program).

The next four statements in the program define the first subplot within the single figure.

```
mpp.subplot(121)
mpp.plot(x1, y1, 'ro')
mpp.title("Plot Using Lists")
mpp.axis([0,7,0,7])
```

The `mpp.subplot(121)` uses a method called **subplot**.

The subplot Method

The subplot method offered by matplotlib.pyplot allows for several graphs or plots to be placed within one figure.

The syntax for the subplot method is:

```
matplotlib.pyplot.subplot(RCL)
```

The “R” is the total number of rows of possible graphs in the subplot formation. The “C” is the total number of columns in the subplot. The “L” is the location of “this” subplot within the defined subplot structure. The R, C, and L are integers.

For example, suppose 6 graphs are to be placed in one figure such that there are three graphs on top (the first row) and three below (the second row). This can be described as a 2 by 3 subplot structure, with 2 rows and 3 columns of graphs. The location of the first graph in the upper left is location 1. In this case, R=2, C=3, and L=1.

In the example program above, `mpp.subplot(121)` specifies that subplots will be located in 1 row and 2 columns, and that the current subplot will be in location 1. The statement, `mpp.plot(x1, y1, 'ro')`, creates a plot using the x values in x1, the corresponding y values in y1, and "red dots" ('ro') to create the plot itself. Figure 9.3 illustrates the results for both subplots.

Example 9.3.2: Polynomial Line Graph with NumPy arrays

In this example, a NumPy array is created to represent the x values. Here,

```
x=np.arange(-100,100,.01)
```

This means that the x values start at -100, end before 100, and are separated by an interval of .01. Here, x = ([-100, -99.99, -99.98, …, 99.98, 99.99])

This array contains 20,000 values. The **length** function, `len(x)`, can be used to determine the length of the array.

The y values are created as a result of a polynomial (quadratic) function of x. Recall that "**" means exponent. This can be read as, y equals 3 times x squared, plus 4 times x, minus 5.

```
y=3*x**2 + 4*x - 5
```

These 20,000 (x,y) pairs are then plotted using the plot function. The axis is specified, as is the title. Figure 9.4 illustrates the results.

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

x=np.arange(-100,100,.01)
y=3*x**2 + 4*x - 5

mpp.plot(x, y, 'r-')
mpp.axis([-5,5,-20,20])
mpp.title("Polynomial Line Graph")
mpp.show()
```

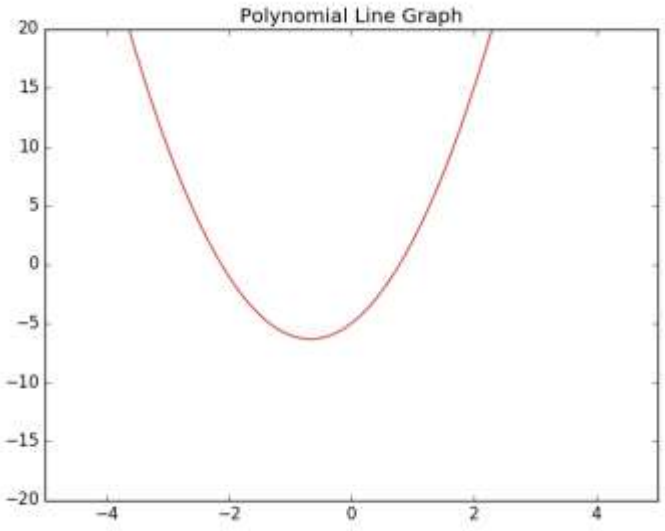


Figure 9.4: The output for example 9.3.2

9.3.2: A Closer look at NumPy Methods

While NumPy offers an enormous number of methods and options, this subsection will explore a few trigonometric functions offered by NumPy, as well as the linspace method for creating arrays of numbers.

Example 9.4.1: Sin, Cos, and Tan in NumPy

```
import numpy

PiOverTwo=numpy.pi/2

PiOverTwo
Out[150]: 1.5707963267948966

numpy.sin(PiOverTwo)
Out[151]: 1.0

numpy.cos(PiOverTwo)
Out[152]: 6.123233995736766e-17

numpy.tan(PiOverTwo)
Out[153]: 16331239353195370.0
```

In Example 9.4.1, NumPy is imported. Unlike most of the examples in this book, this example does not give NumPy a “short name” like np. Recall that giving an import a short name is

optional. Once NumPy is imported, access to sin, cos, and tan, as well as pi are gained. The variable, `PiOverTwo` is π/2. The sin(π/2) is 1, as shown in statements,

```
numpy.sin(PiOverTwo)
Out[151]: 1.0
```

The cos(π/2) is 0, but here, because of the decimal rounding, the results is shown as an extremely small value in scientific notation, where the `e-17` represents the exponent of -17. Written in standard notation, this value is `.00000000000000006123233995736766`. By using the **round** function and rounding to 3 decimal places, the value comes out as the expected 0.

```
numpy.cos(PiOverTwo)
Out[152]: 6.123233995736766e-17
round(numpy.cos(PiOverTwo),3)
Out[154]: 0.0
```

The tan(π/2) is infinity, but here, because of the finite representation in memory of float values, it is not possible for the computer to offer this result. That being said, there are work-arounds such as creating a decision structure (if/else) that tests to see if value exceeds a given number.

```
numpy.tan(PiOverTwo)
Out[153]: 16331239353195370.0
```

Example 9.4.2: Linspace in NumPy

The **linspace** method is similar to the arange method in that it will create an array of values.

The syntax for linspace:

```
import numpy as np
np.linspace(start,stop, numofvalues, endpoint=True,
retstep=False, dtype=none)
```

The `start` parameter is the starting value in the array. It must be numerical.

The `stop` parameter is the ending value in the array. It must be numerical. If the `endpoint` parameter in the linspace method is set to True, the array of values will include the value stop. If `endpoint` is set to False, it will not.

The `numofvalues` parameter is a non-negative integer and specifies the number of digits to have in the array. It is optional and the default is 50.

The `retstep` is an optional **Boolean** parameter. If set to True, it will return the step (separation between the numbers in the array), along with the array itself. Recall that Boolean is True or False.

The `dtype` parameter is the type of the array. This parameter is optional and if omitted, the type of the array will be inferred from the start and stop values.

Example of linspace:

```
np.linspace(1,20,5,retstep=True)
Out[159]: (array([  1.  ,   5.75,  10.5 ,  15.25,  20.  ]),
4.75)
```

Here, the array starts at 1, ends at and includes 20, and the step is returned as 4.75 because retstep is set to True. Note that the 4.75 is the distance or step between each two values in the array.

The key difference between **arange** and **linspace** is that linspace allows for the specification of the number of values in the array (which will imply the step) and arange specifies the step (which will imply the number of values in the array.)

Example 9.4.3: Trigonometry Line Graph Subplots using linspace and arange

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

mpp.figure(1)
## Using arange arrays for x and y
x1=np.arange(-10000,10000,.01)
y1=np.sin(x1)

## Subplot 1 left This is a 1 row, 2 col subplot
mpp.subplot(121)
mpp.plot(x1, y1, 'y-')
mpp.axis([-10,10,-5,5])
mpp.title("Example Sine Function \n Using arange")

## Using linspace arrays for x and y
x2=np.linspace(-10,10,30)
y2=x2**3 - 4*x2 + 5

## Subplot 2 right
mpp.subplot(122)
mpp.plot(x2, y2, 'b--')
mpp.title("Polynomial Plot \n Using Linspace")
mpp.axis([-10,10,-20,20])

#Show the plot
mpp.show()
```

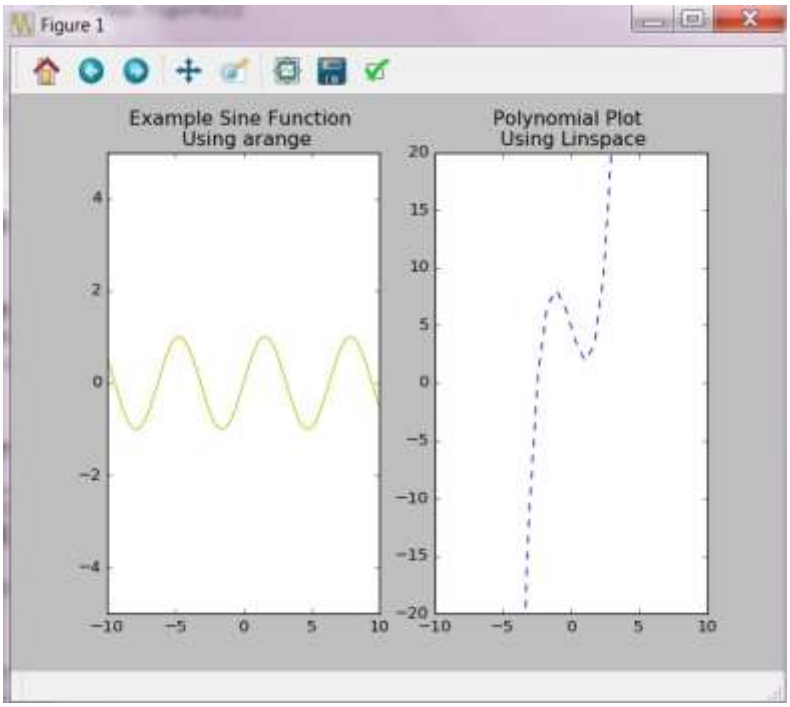


Figure 9.5: Illustration of Example Program 9.4.3

9.5: Bar Graphs and Pie Graphs

Python and MatPlotLib offer several data visualization tools. This section will discuss the bar graph and the pie graph.

Both the bar graph and the pie graph require some pre-processing of the data before they can be used. For example, quantitative and continuous data cannot be successfully visualized using a bar graph until it is categorized or classified. In the case where the data is already categorized, it can be used directly.

9.5.1: The Bar Graph

The syntax of the bar graph is:

```
matplotlib.pyplot.bar(left, height, width=0.8, facecolor,
edgecolor, align)
```

The `left` parameter is the x coordinate for the left edge of each bar in the bar graph.

The `height` parameter is the height of each bar on the y axis.

The `width` parameter is the width of each bar in the bar graph. This is optional and the default is .8

The `facecolor` parameter is optional and will define the color of the bars in the bar graph.

The `edgecolor` parameter is optional and will define the color of the border around the bars in the bar graph.

The `align` parameter is optional and will define the alignment of the left edge of each bar as center, left, or right.

There are many other optional parameters for bar graphs.

The **xticks** method allows the base of each bar to be labeled and to contain a tick mark.

Example 9.5.1: Bar Graph

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

### BAR
xvals=[1,2,3,4,5]
grades=["A", "B", "C", "D", "F"]
yvals=[.30,.40,.15,.10,.05]

mpp.bar(xvals, yvals, facecolor='#9999ff',
edgecolor='white', align='center')

mpp.xticks(xvals, grades)
name="Bar Graph 1"

mpp.title(name)
mpp.show()
```

Figure 9.6: Example Bar Graph Output from Example 9.5.1

Discussion of Example 9.5.1:

To create this example bar graph, grade data was collected and then processed. The percentage of each letter grade was determined. The `xvals` list represents the 5 bars that will be created (for A, B, C, D, and F). The `yvals` list contains the height of each bar. The `grades` list is the label for each bar. Specifically, the data that was utilized was first categorized into: 30% A, 40% B, 15% C, 10% D, and 5% F.

```
xvals=[1,2,3,4,5]
grades=["A", "B", "C", "D", "F"]
yvals=[.30,.40,.15,.10,.05]
```

9.5.2: The Pie Graph

The pie graph has many optional parameters. This noted syntax includes the required parameters, and a few extra common parameter options.

```
matplotlib.pyplot.pie(values, explode, labels, colors,
autopct, shadow=True, startangle)
```

The `values` parameter is required and will contain the percentage or decimal value of each slice in the pie.

The `explode` parameter (optional) is the fraction of the radius to offset each wedge.

The `labels` parameter (optional) is a list of strings that will label each slice of the pie.

The `colors` parameter (optional) will color each slice of the pie.

The `autopct` parameter (optional) will format the percentage or decimal value in each slice.

The `shadow` parameter (optional) if set to True will create a shadow effect on the pie graph.

The `startangle` parameter (optional) will rotate the starting position of the pie by the given angle (clockwise from the x axis).

Example 9.5.2: Example Pie Graph

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

```
##PIE PLOT
slices=["chocolate", "vanilla", "pecan","coffee",
"lemon"]
values=[.56,.12,.08,.14,.10]
colors=['brown','white','tan','black','yellow']

explode=(.1,0,0,0,0)

mpp.pie(values, explode=explode, labels=slices,
colors=colors,autopct='%1.1f%%', shadow=True,
startangle=160)

mpp.axis("equal")
mpp.show()
```

Output

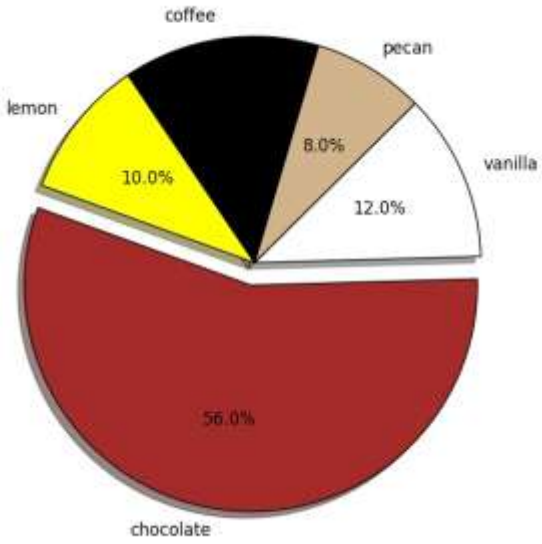


Figure 9.7: Pie Graph Output from Example 9.5.2

Discussion of Example 9.5.2:

From the example code above, consider the four statements:

```
slices=["chocolate", "vanilla", "pecan","coffee", "lemon"]
values=[.56,.12,.08,.14,.10]
colors=['brown','white','tan','black','yellow']
explode=(.1,0,0,0,0)
```

Here, the parameters of the pie method are being defined. The `slices` list will represent the names in each slice of the pie graph. The `values` list will represent the percentage values inside each pie slice. The `colors` will designate the color each pie slice (clockwise). The `explode` sequence notes that the first slice come out by 10% and all others remain in the normal position.

The next statement,

```
mpp.pie(values, explode=explode, labels=slices,
colors=colors,autopct='%1.1f%%', shadow=True, startangle=160)
```

utilizes the defined lists to build the pie graph. Notice that the `explode` parameter is set equal to the `explode` list defined above. Similarly, the `labels` parameter is set equal to the `slices` list that defines each pie slice label. The `colors` parameter is set to the `colors` list defined above, and the `autopct` is set to `%1.1f%%`. The `autopct` defines the format of the numerical values that will be contained in each pie slice. The `%1.1f%%` specifies that there will be one decimal place, that the number is a float (f), and that the value will contain the % symbol.

9.5.3: Scatterplots

A scatterplot is a visual representation of the relationship between two variables and their paired datasets.

The basic syntax for a scatterplot is:

```
matplotlib.pyplot.scatter(xvalues, yvalues)
```

There are several optional parameters.

Example 9.5.3: Creating a Scatteplot

```
import nupPy as np
import matplotlib.pyplot as mpp

##SCATTER
NumCigs=[18.20,25.82,18.24,28.24,31.10,33.60,40.46]
LungCancer=[17.05,19.80,15.98,22.07,22.83,24.55,27.27]

mpp.scatter(NumCigs,LungCancer)
mpp.axis([0,50,0,50])
mpp.title("Relationship Between Smoking and Lung Cancer")
mpp.show()
```

```
#Data Ref:
#http://lib.stat.cmu.edu/DASL/Datafiles/cigcancerdat.html
```

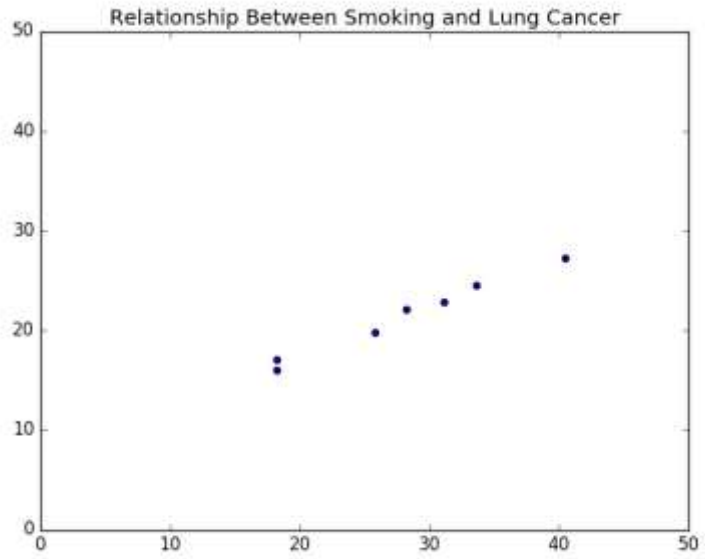


Figure 9.8: Illustration of the Scatterplot in Example 9.5.3

9.5.4: Subplots, Decision Structures, and Loops with Graphics

Subplots

Within a given figure, it is feasible to have several plots. When several plots appear within one figure, each plot within that figure is called a **subplot**. The **shape** of the set of subplots can be thought of as a matrix of the plots, with R rows and C columns. Figure 9.9 illustrates a figure with 6 subplots. Specifically, this example illustration has 2 rows and 3 columns.

To create a subplot, the subplot method is used, where R is the number of total rows of subplots, C is total number of columns of subplots, and L is the location of the current plot. Location is counted from the upper left to the lower right. For example, *mpp.subplot(234)* has 2 rows, 3 columns, and the current plot is in location 4 (of the 6 subplots).

The Syntax for subplots:

```
matplotlib.pyplot.subplot(RCL)
```

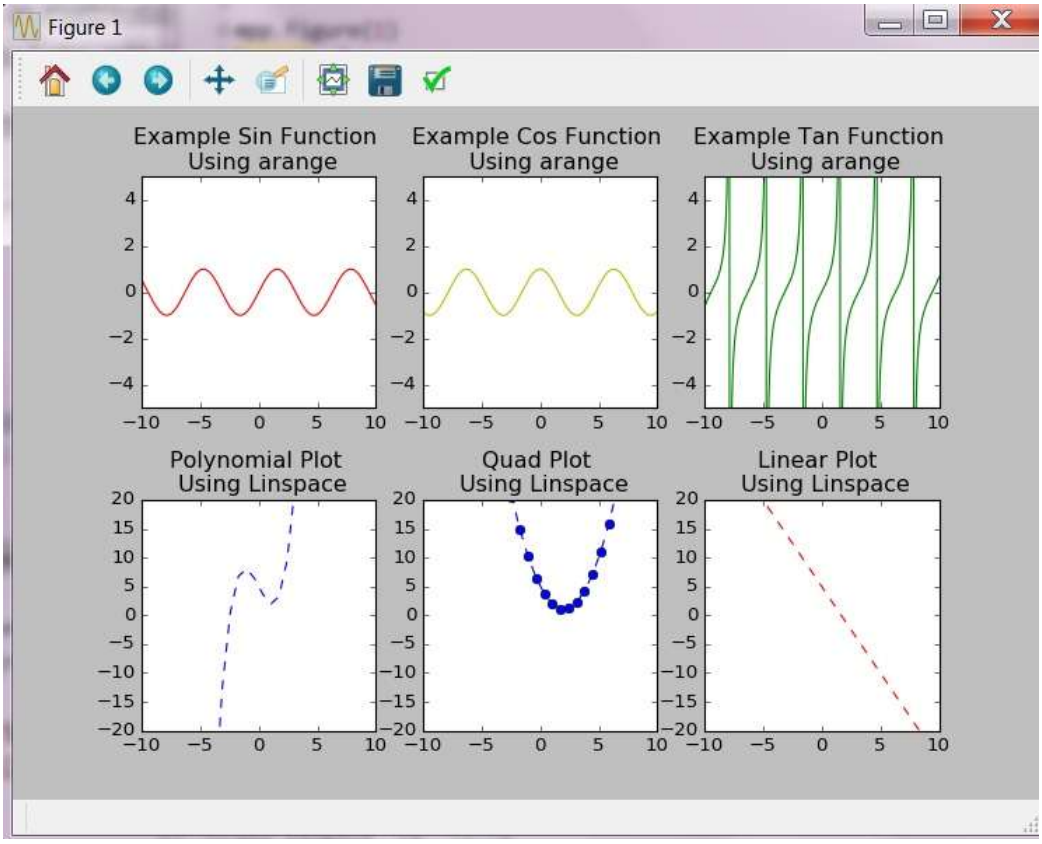


Figure 9.9: Illustration of Subplots with matrix/shape of 2 rows and 3 columns

The titles of subplots can overlap with neighboring plots. To adjust the look of the overall figure, the **subplots_adjust** method can be used to add space between subplots.

The subplots_adjust syntax is:

```
mpp.subplots_adjust(hspace)
```

The parameter, `hspace`, specifies the extra space to be placed under each subplot. In Figure 9.9, the hspace is set to .4.

The following program contains the code that created the plots in Figure 9.9.

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

mpp.figure(1)
hspace=.4
mpp.subplots_adjust(hspace=hspace )
## Using arange arrays for x and y
x1=np.arange(-10000,10000,.01)
y1=np.sin(x1)
```

```
## Subplot 1 left This is a 1 row, 2 col subplot
mpp.subplot(231)
mpp.plot(x1, y1, 'r-')
mpp.axis([-10,10,-5,5])
mpp.title("Example Sin Function \n Using arange")

## Using arange arrays for x and y
x1=np.arange(-10000,10000,.01)
y1=np.cos(x1)

## Subplot 1 left This is a 1 row, 2 col subplot
mpp.subplot(232)
mpp.plot(x1, y1, 'y-')
mpp.axis([-10,10,-5,5])
mpp.title("Example Cos Function \n Using arange")

## Using arange arrays for x and y
x1=np.arange(-10000,10000,.01)
y1=np.tan(x1)

## Subplot 1 left This is a 1 row, 2 col subplot
mpp.subplot(233)
mpp.plot(x1, y1, 'g-')
mpp.axis([-10,10,-5,5])
mpp.title("Example Tan Function \n Using arange")

## Using linspace arrays for x and y
x2=np.linspace(-10,10,30)
y2=x2**3 - 4*x2 + 5

## Subplot 2 right
mpp.subplot(234)
mpp.plot(x2, y2, 'b--')
mpp.title("Polynomial Plot \n Using Linspace")
mpp.axis([-10,10,-20,20])

## Using linspace arrays for x and y
x2=np.linspace(-10,10,30)
y2=x2**2 - 4*x2 + 5

## Subplot 2 right
mpp.subplot(235)
mpp.plot(x2, y2, 'o--')
mpp.title("Quad Plot \n Using Linspace")
mpp.axis([-10,10,-20,20])
```

```
## Using linspace arrays for x and y
x2=np.linspace(-10,10,30)
y2=x2 - 4*x2 + 5

## Subplot 2 right
mpp.subplot(236)
mpp.plot(x2, y2, 'r--')
mpp.title("Linear Plot \n Using Linspace")
mpp.axis([-10,10,-20,20])



#Show the plot
mpp.show()
```

Decision Structures and Loops with Graphics

Decision structures, such as the if/elif/else, and loops, such as the for/in loop can be used in conjunction with visualization to offer versatility. The following example uses a for/in loop to create 4 different polynomial graphs. It also uses the if/elif/else decision structure to determine the line shape and color for each subplot. Functions can also be utilized in combination with graphics to create more robust programs.

Example 9.5.4: Graphing with Loops and Decisions

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

def main():
    x=np.arange(-20,20,1)

    for i in [1,2,3,4]:
        y=x**i

        if i==1:
            ltype='ro'
            splot=221
        elif i==2:
            ltype='b^'
            splot=222
        elif i==3:
            ltype='g-'
            splot=223
        else:
```

```
            ltype='yo'
            splot=224
        MakePlot(i,x,y,ltype,splot)


def MakePlot(plotnum,xvals,yvals,linetype='ro',sp=221):
    mpp.subplot(sp)
    mpp.plot(xvals, yvals, linetype)
    name="Plot Number "+ str(plotnum)
    mpp.title(name)
    mpp.axis([-10,10,-10,10])
    #Show the plot
    mpp.show()

main()
```

Output:

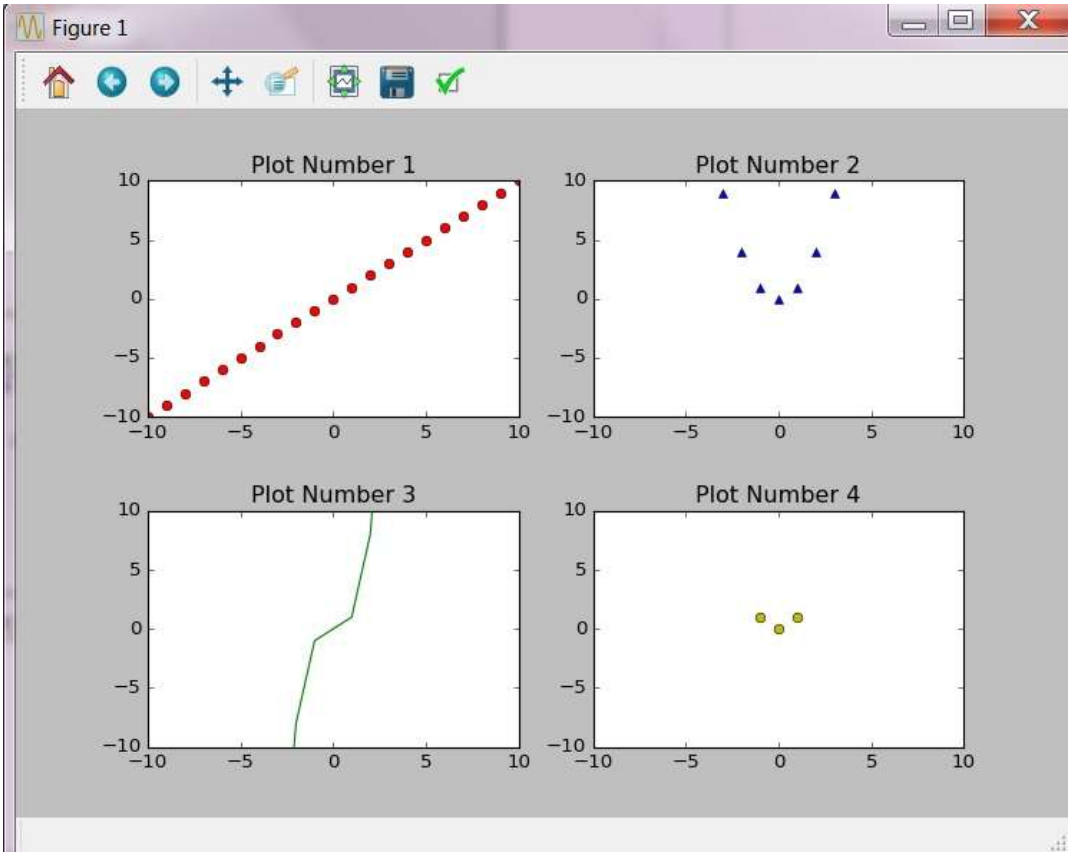


Figure 9.10: Graphic output from Program in Example 9.5.4

Review of the Example 9.5.4 by Line of Code:

```
1.  import numpy as np
2.  import matplotlib.pyplot as mpp

3.  def main():
4.      x=np.arange(-20,20,1)

5.      for i in [1,2,3,4]:
6.          y=x**i

7.          if i==1:
8.              ltype='ro'
9.              splot=221
10.         elif i==2:
11.             ltype='b^'
12.             splot=222
13.         elif i==3:
14.             ltype='g-'
15.             splot=223
16.         else:
17.             ltype='yo'
18.             splot=224
19.         MakePlot(i,x,y,ltype,splot)


20. def MakePlot(plotnum,xvals,yvals,linetype='ro',sp=221):
21.     mpp.subplot(sp)
22.     mpp.plot(xvals, yvals, linetype)
23.     name="Plot Number "+ str(plotnum)
24.     mpp.title(name)
25.     mpp.axis([-10,10,-10,10])
26.     #Show the plot
27.     mpp.show()

28. main()
```

Lines 1 and 2:

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

Line 1 imports the package called numpy, and gives it a "short name" of "np". Line 2 imports matplotlib.pyplot using short name "mpp".

Line 3

Line 3 defines the main function of the program. The main function has no parameters.

Line 4

Line 4 creates the x values using the arange method. The x values are ([-20, -19, …,, 18, 19]). Recall that arange does not include the last or stopping value.

Lines 5 – 19:

Lines 5 – 19 are contained within a *for loop*. The for loop sets the variable "i" equal to 1, 2, 3, and then 4. For each "i" value, line 6 creates the y values as x^i. Therefore, when i = 1, y is x, when i = 2, y = x^2, when i=3, y=x^3, and when i=4, y=x^4.

Lines 7 – 18 are contained in the for/in loop and create a decision structure using the if/elif/else construct. As the for/in loop updates the value of "i", a different decision is made. For example, when i = 1, the first if statement is entered (lines 7 – 9). In this case, the ltype (which is the type of line) is set to "ro" which is red dots. The splot (which is the subplot) is set to 221, which is 2 rows, 2 columns, and location 1.

Line 19 creates the plot by calling the function called `MakePlot` with all the needed parameters, such as the i value, the x values, the y values, the line type (as ltype), and the subplot values (as splot).

Lines 20 – 27

Lines 20 – 27 defines the `MakePlot` function.

```
def MakePlot(plotnum,xvals,yvals,linetype='ro',sp=221)
```

Notice that this code creates one figure with 4 subplots. The for loop on line 5 loops through each of the 4 subplots and defines them. For example, when i = 1, the line type is set to "ro" and the subplot (as splot) is set to 221. Within the for loop on line 6, the y values are created. On line 4 above the for loop, the x values are created

The `MakePlot` function is called (on line 19) with the parameters needed to create that subplot. This is done 4 times because the call to `MakePlot` is inside of the for/in loop.

Line 21 takes the parameter "sp" which stands for subplot and sets the subplot. When i = 1, for example, sp will be set to splot, which is 221.

Line 22 calls the plot method with the x values, y values and line type for that subplot.

Line 23 uses string concatentation to name the plot depending on the value of the plot number (which is the value of i).

Line 28 calls the main function and starts the program.

Example 9.5.5: Pulling the Concepts Together

Python, in conjunction with NumPy and MatPlotLib, can create a large variety of graphs. This example will pull together several graphics examples, as well as functions, loops, and decision structures. The following example program will generate a variety of outputs. Figures 9.11 – 9.14 will illustrate different outputs. To practice, type in and run the program.

```
# GraphingExamples.py
# Ami Gates

import numpy as np
import matplotlib.pyplot as mpp

def main():

    plottype=input("Enter plot type. scatter, line, pie, or bar: ")

    ## subplot=220: Initializes a 2 row and 2 column subplot figure
    ## 221: place a subplot is the first position of the 2X2 subplot

    subplot=220
    ## Recall that "ro" is red dots.
    linetypes=["ro","b^","g-","yo"]

    for i in [1,2,3,4]:
        subplot=subplot+1
        PrepPlot(plottype[0],i,linetypes[i-1],subplot)

 ###End of Main


def MakePlot(i,xvals,yvals, ptype, linetype='ro',sp=221):
    ## "i" is the subplot number, sp is the subplot location

    mpp.subplot(sp)

    if ptype[0]=="s"or ptype[0]=="S":
        mpp.scatter(xvals,yvals)
        mpp.axis([-20,20,-20,20])

    elif ptype[0]=="l"or ptype[0]=="L":
        mpp.plot(xvals, yvals, linetype)
        mpp.axis([-20,20,-1000,1000])

    elif ptype[0]=="p" or ptype[0]=="P":
```

```
        #print("pie")
        mpp.pie(xvals)
        mpp.axis("equal")

    else:
        #print("bar")
        mpp.bar(xvals, +yvals, facecolor='#9999ff', edgecolor='white')

    name="Plot Number "+ str(i)
    mpp.title(name)

    #Show the plot
    mpp.show()
### end of MakePlot


def PrepPlot(pt,i,linet,splot):
    ## Here, pt is the plottype, i is the subplot, linet is the
linetype, splot is the
    ## subplot location (such as 221 for the first one)

    if pt=="s" or pt=="S":
        #Scatterplot
        n = 1024
        x = np.random.normal(0,i,n)
        y = np.random.normal(0,i,n)

    if pt=="l" or pt=="L":
        #Lineplot
        x = arange(-200,200,i)
        y = x**i

    if pt=="b" or pt=="B":
        #barchart
        n=3*i
        x = arange(n)
        y = x * np.random.uniform(0,1.0,n)

    if pt=="p" or pt=="P":
        #pie
        n=3*i
        x=np.random.uniform(0,3*i,n)
        y = 0

    MakePlot(i,x,y,pt,linet,splot)
########end of PrepPlot

main()
```

Outputs for the above program:

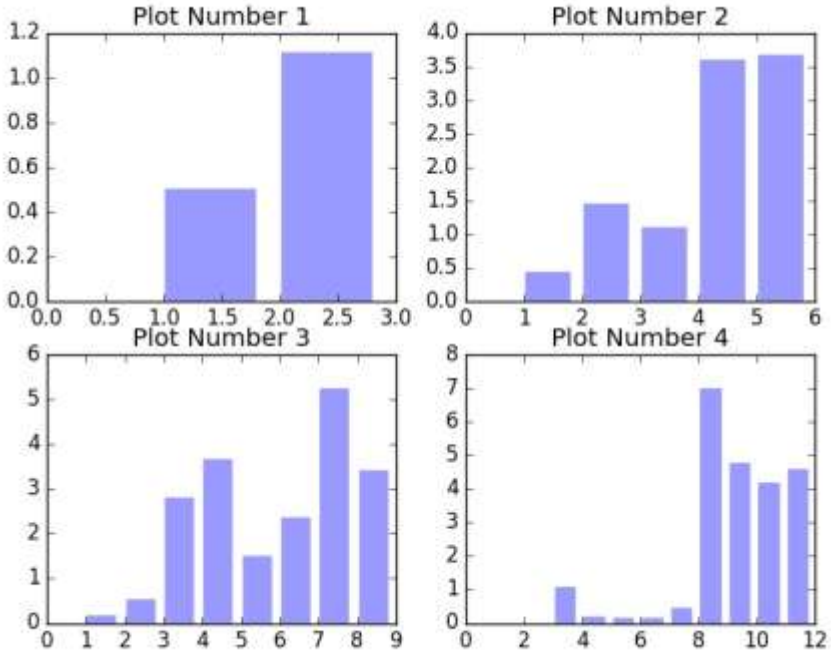


Figure 9.11: Program result from input: Bar

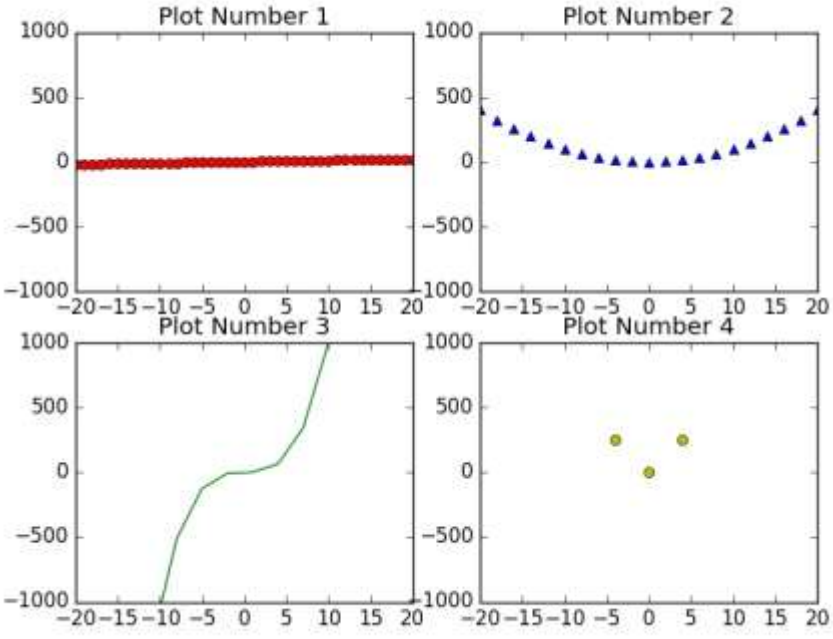


Figure 9.12: Program result from input: Line

Figure 9.13: Program result from input: Pie

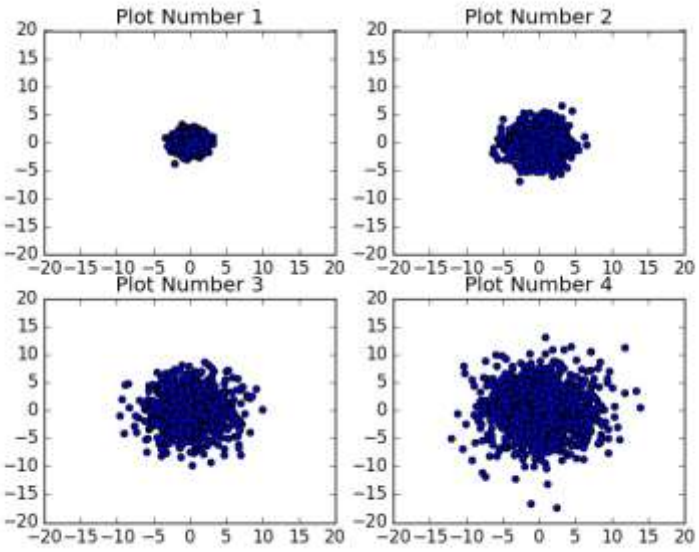


Figure 9.14: Program result from input: Scatter

Discussion of Program Example 9.5.5, `GraphingExamples.py`

The GraphingExamples.py program is composed of three functions, `main`, `MakePlot`, and `PrepPlot`.

The `main` function is called first at the very bottom of the program and it controls the flow of the overall program. The `main` function does not require or return any parameters or values. Inside of the `main` function definition, the user is asked to choose a graph type. The four choices are scatterplot, line graph, bar graph, or pie graph. The information that the user enters as input (types in) is collected and stored in the variable called, `plottype`. Notice that in the program, `plottype[0]` is used. Because `plottype` is a string, `plottype[0]` is the first character in the string (word) that the user enters. For example, if the user enters "scatterplot", then `plottype[0]` will be "s". This is a valuable method because the user might enter, scatterplot, scatter, Scatter plot, etc. This same logic follows for user inputs such as bar, bar chart, bar graph, line, line plot, pie, pie graph, etc.

Next, the variable called `subplot` is initialized to 220. Recall that Python can create subplots within a plot or Figure area. When defining a subplot, first the total number of rows and columns being used by subplots is specified, followed by the location of each subplot. For example, the value 221 specifies 2 rows and 2 columns, and notes that the subplot in question (221) goes in location 1 of that collection of subplots. This can be seen in the plot images above, as each plot is numbered. Plot Number 1 is 221, Plot Number 2 is 222, Plot Number 3 is 223, and Plot Number 4 is 224. This concept extends to any number of rows and columns so that 325 would be 3 rows and 2 columns of subplots, with the subplot (325) in location 5 of that collection.

Next, inside of `main`, a `for/in` loop repeats four times with the variable "i" equal to 1, 2, 3 and 4. Within the `for/in` loop, the subplot is updated. For example, when i = 1, the subplot is 221, and when i = 4 the subplot is 224. This allows the loop to control which plot location to use. In addition, inside of the for/in loop in main, the function called `PrepPlot` is called. Because `PrepPlot` is inside of the for/in loop, it will be called four times, with corresponding parameter values.

The `PrepPlot` function definition specifies four parameters: `pt, i, linet`, and `splot`. The `pt` parameter represents the plot type. The plot type options are scatter, bar, line, and pie. The next parameter, `i`, is the subplot number. The third parameter, `linet`, is the type of the line, such as "ro" for red dots, etc. The fourth parameter is `splot`, which is the location of the subplot (such as 221 or 222 in this case). When `PrepPlot` is called, it expects four values in this order.

Inside of the function definition for `PrepPlot`, there are a series of `if` statements. Each `if` statement determines the plot type (pt) that the user entered. If the user entered scatterplot, then the plot type (pt) will equal "s" or "S". Why is it best to include lowercase and uppercase "s"? Because there is no method for forcing the user to be case sensitive. This method will work whether the user enters a capitalized word or not.

Following this logic, the `PrepPlot` function determines which graph type is expected, and then based on the graph type, creates the corresponding "x" and "y" values needed to offer an example of that type of graph. Once the x and y values are created, the function `MakePlot` is called from *within* `PrepPlot`. This is a great example of a function being called from within another function other than main.

The `MakePlot` is defined and called with six parameter values. The last two of the six parameters have **default** values associated with them and so are not required. The `MakePlot` function parameters include `i`, the subplot number, `xvals`, the x values for the graph, `yvals`, the y values for the graph, `ptype`, which is the plot type (such as scatter, bar, line, or pie), the `linetype` (such as ro or g-), and `sp` (which is the location of the subplot such as 221 or 222).

Inside of the `MakePlot` function definition, the subplot method, `mpp.subplot(sp)`, creates a subplot in the location specified by `sp` (in our example this will be 221, 222, 223, or 224). Next, into that subplot location, a graph is created based on the plot type (ptype) requested by the user.

For example, the following few statements tests to see if ptype[0] is "s" or "S". If the first character of ptype is an upper or lowercase s, then this will be a scatterplot. In this case, note that `mpp.scatter` is called with the xvals and yvals. In addition, the mpp axis method is called to specify that the x axis (in this case) will range from -20 to 20, and the y axis will also range from -20 to 20.

```
if ptype[0]=="s"or ptype[0]=="S":
        mpp.scatter(xvals,yvals)
        mpp.axis([-20,20,-20,20])
```

The `MakePlot` function contains a series of if/elif/else statements that create a decision structure. This structure determines the plot type requested by the user and creates that specific graph in the subplot area.

The last three statements in the `MakePlot` function are:

```
name="Plot Number "+ str(i)
    mpp.title(name)

    #Show the plot
    mpp.show()
```

Here, the `name` that appears above each subplot in the images above is created by using the "+" (concatenation) option. The words, "*Plot Number* " are connected (concatenated) to the string version of the plot number. Recall that "i" is the plot number (1, 2, 3 or 4). However, the variable "i" is a number (integer type) and so cannot be concatenated to a string. To work around this, the data type of "i" is first cast (converted) into a string using the "str" function, `str(i)`. Then the string typed "i" can be concatenated onto "Plot Number ". This method results in the titles on top of each subplot above, such as "Plot Number 1" and "Plot Number 2", etc.

The `mpp.title()` is a method that creates and places that title onto the subplot. Finally, the `mpp.show()` method renders (make visible) the subplot. The program above also uses a method called "`random`" which is part of the NumPy library.

The random method

The `random` method can generate random values from a given distribution. For example, the statement,

```
x = numpy.random.normal(0,1,25),
```

will create an array of 25 randomly generated values from a normal distribution with a mean of 0 and a standard deviation of 1.

Similarly, the statement,

```
y=numpy.random.uniform(0,10,20),
```

will create an array of 20 values that were randomly generated from a uniform distribution with a minimum of 0 and a maximum of 10.

Recall that in the program above, the package NumPy is imported with the "nickname" of np. For this reason it is possible to call random using the nickname. For example, the statement,

```
np.random.normal(60,5,15)
```

will create an array of 15 values that were randomly generated from a normal distribution with a mean of 60 and a standard deviation of 5.

Example 9.5.6: A Last Look at Basic Graphs, Random, and Subplotting

This final example in this section will pull together the concepts covered thus far for graphing. It is recommended that you type in, save, run, and make updates to this program.

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

mpp.subplot(221)
n = 1024
x = np.random.normal(0,4,n)
y = np.random.normal(0,4,n)
mpp.scatter(x,y)
mpp.axis([-20,20,-20,20])
mpp.title("2D Random Normal Plot\n")
```

```
mpp.show()

#-Subplot 2
mpp.subplot(222)
slices=["chocolate", "vanilla", "pecan","mint",
"lemon"]
values=[.56,.12,.08,.14,.10]
colors=['brown','white','tan','green','yellow']
explode=(.1,0,0,0,0)
mpp.pie(values, explode=explode, labels=slices,
colors=colors, autopct='%1i%%', shadow=True,
startangle=160)
mpp.axis("equal")
mpp.title("Flavor Preference\n")
mpp.show()

#- subplot 3
mpp.subplot(223)
xvals=[1,2,3,4,5]
grades=["A", "B", "C", "D", "F"]
yvals=[.30,.40,.15,.10,.05]
mpp.bar(xvals, yvals, facecolor='#0099ff',
edgecolor='white', align='center')
mpp.xticks(xvals, grades)
name="\n\nDistribution of Letter Grades"
mpp.title(name)
mpp.show()

#- subplot 4
mpp.subplot(224)
x=np.arange(-10000,10000,.01)
y=np.cos(x)
mpp.plot(x, y, 'y-')
mpp.axis([-10,10,-5,5])
mpp.title("\n\nExample Cos Function Plot")
mpp.show()
```

Figure 9.15 illustrates the output for this program.

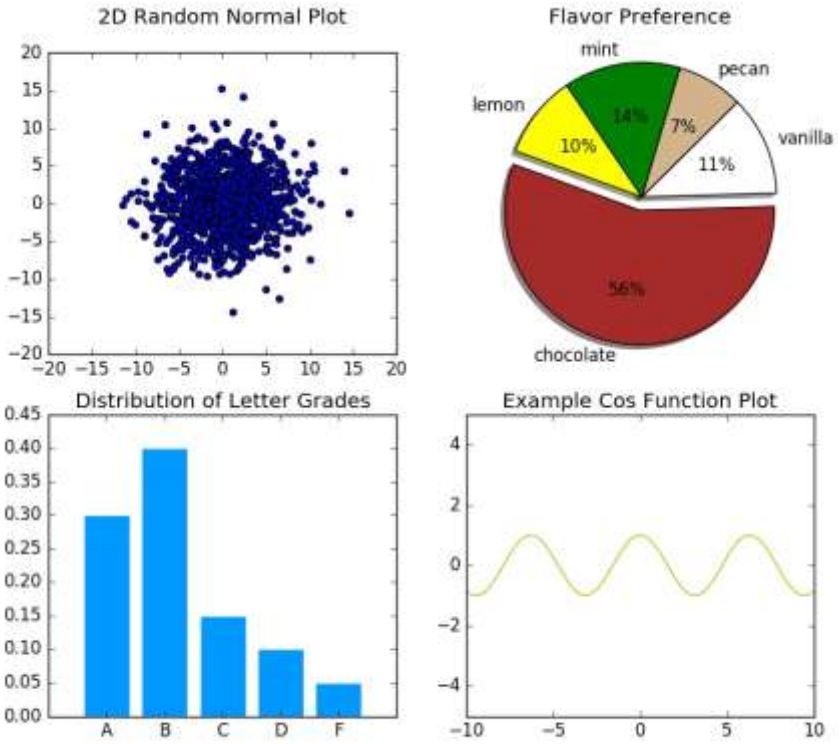


Figure 9.15: Output for example 9.5.6

9.6: Further Data Visualization Options and a Case Study

Python, via several data visualization tools and packages offers many graphics options, including interactive data visualization options including Bokeh (http://bokeh.pydata.org/en/latest/), plotly (https://plot.ly/python/), geoplotlib with pyglet (https://github.com/andrea-cuttone/geoplotlib), ggplot via R (https://pypi.python.org/pypi/ggplot), pandas (http://pandas.pydata.org/), and many others.

While this text does not focus on data visualization, the next example will show the use of NumPy, MatPlotLib.pyplot, and mpl_toolkits.mplot3d, Axes3D, to create a 3D graph.

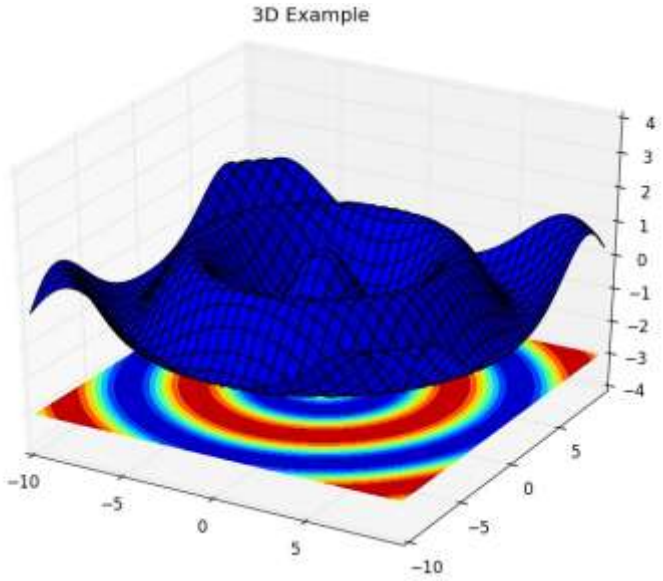
Example 9.6.1: 3D Graphics in Python

The following is an example program of a 3D plot using NumPy, MatPlotLib.pyplot, and mpl_toolkits.mplot3d, Axes3D. Review the following small program.

```
import numpy as np
import matplotlib.pyplot as mpp
from mpl_toolkits.mplot3d import Axes3D

new_fig = mpp.figure()
```

```
axis = Axes3D(new_fig)

x = np.arange(-10, 10, 0.3)
y = np.arange(-10, 10, 0.3)

x, y = np.meshgrid(x, y)

r = np.sqrt(x**2 + y**2)
z = np.cos(r)

axis.plot_surface(x, y, z, rstride=2, cstride=2)
axis.contourf(x, y, z, zdir='z', offset=-3)
axis.set_zlim(-4, 4)
mpp.title("3D Example\n")
mpp.show()
```

The output of this small example program is:



9.6.2: Case Study: Combining Files, Data, and Graphics

This final example in Chapter 9 will illustrate most of the concepts and constructs covered thus far. Example 9.6.2 will combine the use of data, file I/O, and graphics via a case study. It is recommended that you review the case study below. Then, plan, design (via flowchart), and write the program to solve the case study. Once completed, review the example code solution. There is always more than one solution to such questions. If your code differs from the given

solution, it is not incorrect. However, comparing your solution with the given solution may add further insight into these concepts.

The Case Study Specifications:

The goal of this Case Study is to utilize many of the constructs and concepts learned in chapters 1 – 9. Create a program that reads a dataset. The dataset can be as large as desired and can have as many variables as desired. However, it is recommend, as a first attempt, to have six to eight variables and about 20-30 lines of data. The data should be "tab delineated", in other words, all variables and values are separated by TAB (\t).

Once the program reads in the data from the dataset, it asks the user to choose two variables from the dataset to evaluate. Keep in mind that users can type in variable names in a variety of ways and the program should be robust and easy to use. Try to guess what a user might do and prepare for it in the code.

Once the user enters the two variables, the program should output at least the following:
1) A Figure with six subplots: A scatteplot of both variables, a bar graph with properly categorized data for one of the variables, two box plots (one for each variable), and two pie graphs that are properly formatted to offer some information about each variable.
2) The mean, median, variance, standard deviation, max, and min for both variables.

A Solution to the Case Study:

The following solution includes a copy of the exact dataset used, the output for the program (including the graphical output), and a copy of the program code.

The Dataset:

The small Data Set used for the Case Study (tab delineated)

Ch9DataSet.txt

Note: Each item in this dataset is separated by tabs (\t)

```
ID   CLA  LET  SEX  AGE  GPA  HRS  FINAL
3461 90.3 A    M    34   3.67 12   97.5
3323 89.1 B    M    21   3.21 10   88.3
1876 99.3 A    F    28   3.44 16   98.9
3965 77.5 C    F    19   3.01 5    78.8
4526 81.9 B    M    28   2.98 13   85.2
1287 56.1 F    F    30   1.15 2    56.7
3368 75.1 C    F    25   2.45 15   77.4
9937 92.2 A    M    35   3.88 13   92.1
8844 72.8 C    F    34   2.89 5    78.3
7856 96.6 A    M    37   3.47 10   93.1
8754 67.1 D    M    26   1.65 5    68.2
```

```
6689 88.4 B     F     31    3.11 14    81.5
9000 83.2 B     F     32    2.98 12    86.3
4587 90.4 A     F     38    3.89 14    90.2
8919 73.2 C     M     34    2.89 7     80.1
2341 78.4 C     M     54    2.66 8     78.2
3465 90.1 A     M     26    3.77 15    93.6
4678 55.2 F     M     67    1.77 3     55.2
4867 98.7 A     F     45    3.56 12    95.6
9834 94.6 A     F     36    3.34 10    98
8712 90.2 A     F     33    3.48 16    89.3
```

The Graphical Output

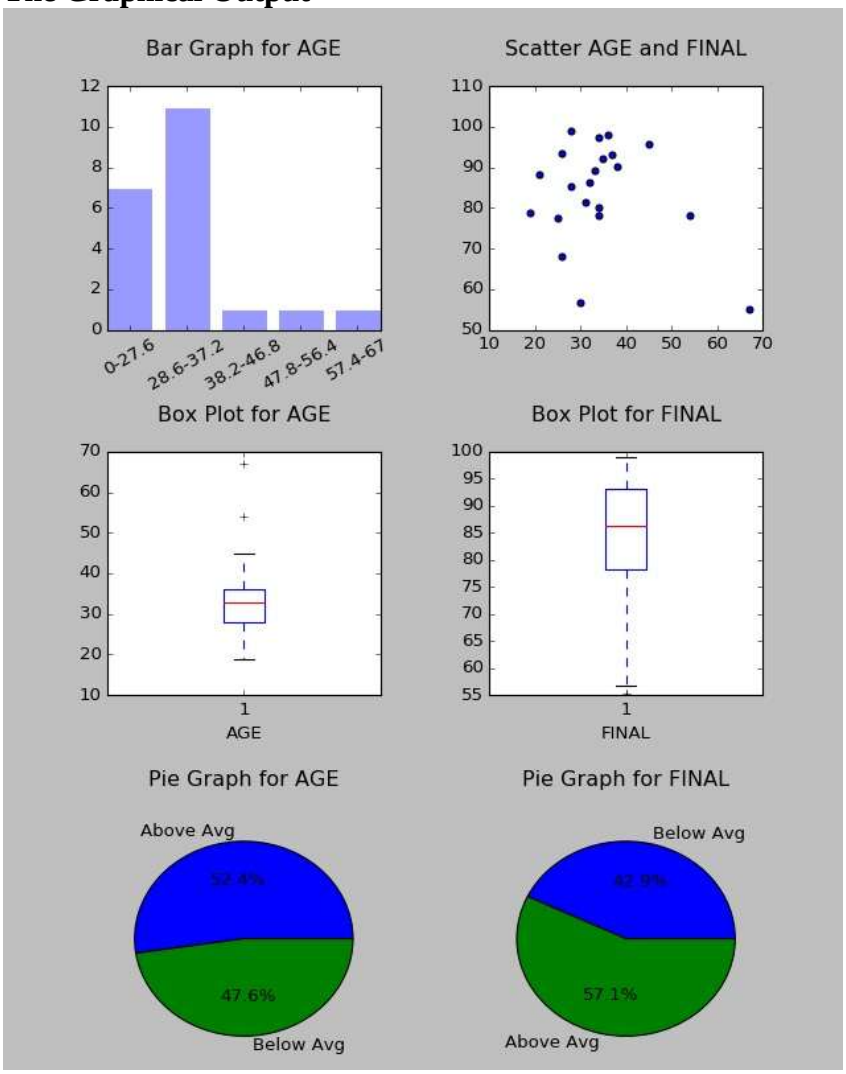


Figure 9.6.1: A possible graphical output for the Case Study Program

Example Program Code Solution for Chapter 9 Case Study:

```
# Chapter9_CaseStudy.py
# Ami Gates
#-----------------------------------------------------
# This program works on dataset Ch9DataSet.txt
# It can be updated to work on any dataset
# The user is asked to enter two numeric
# variables from the dataset
# The program will create a scatterplot for both vars
# a bar graph for var 1, boxplots for both vars
# and pie graphs for both vars as a Figure with 6 subplots
# The program will also output summary stats for both vars
#-------------------------------------------------------------

import numpy as np
import matplotlib.pyplot as mpp

def main():

    ## Communicate with User
    print("Welcome\n")
    print("This program will allow you to choose any two variables and \n")
    print("it will print out basic summary statistics and graphs, \n")
    print("for the two variables selected. \n")
    print("_____________________________________________________________\n")
    print("The data set used for this program is called, Ch9DataSet.txt")
    print("The VARIABLES in the dataset are:  ID, CLA \(avg classgrade\)"),
    print("LET \(for final class letter grade\), SEX, AGE, \n")
    print("GPA, HRS \(for hours of study per week\)")
    print("and FINAL \(for final exam score\)")
    print("_____________________________________________________________\n")

    ## This dictionary data structure will key the var names with numbers
    ## Notice that the caps and lwer case options are included
    ## If the user types in any word starting with I or i, such as ID
    ## This dict maps it to variable 0 (the first) in the dataset.
    VarNameDict={"I":0, "i":0, "C":1, "c":1, "L":2, "l":2,
                 "S":3, "s":3, "A":4, "a":4, "G":5,"g":5,
                 "H":6, "h":6, "F":7,"f":7}
    ## This set contains the first letter of qualitative var
    #3 The user cannot choose a qual var
    QualVarSet=("I", "i", "l","L","S","s")

    #print(VarNameDict)

    ## Get user input
    UserInVar1=input("Enter the name of the first numerical variable to include in the
analysis: ")
    while UserInVar1[0] in QualVarSet:
        print("\nPlease select a numerical variable for analysis\n")
        UserInVar1=input("Enter the name of the first numerical variable to include in
the analysis: ")


    UserInVar2=input("Enter the name of the second numerical variable to include in
the analysis: ")
    while UserInVar2[0] in QualVarSet:
        print("\nPlease select a numerical variable for analysis\n")
        UserInVar2=input("Enter the name of the first numerical variable to include in
the analysis: ")

    ## Get all of the data and place it into variable TheData
```

```
    TheData=GetData("Ch9DataSet.txt")

    # print all the data to confirm. Note that because this data
    # came from Excel, each item in the dataset is separated by a TAB
    # The symbol for TAB is \t
    ##print(TheData)

    #Determine the number of lines of data that are in the
    #dataset. Note that line 1 of this example dataset are variable names.
    numlines=len(TheData)
    ##print(numlines)

    #print the variable names only. Split the names using the TAB (\t)
    #place the result in the varnames list
    #--varnames=TheData[0].rsplit("\t")
    #--print(varnames)


    # Call the function that prints summary stats for the two variables of choice

var1_list,var2_list=EvaluateData(UserInVar1,UserInVar2,VarNameDict,numlines,TheData)

    # Call function to make the graphs
    CreateGraphs(var1_list,var2_list,UserInVar1,UserInVar2)



#---------------------------------
def GetData(filename):
        fp=open(filename, "r")
        AllData=fp.readlines()
        fp.close()
        return AllData


#-----------------------------------------------------
def CreateGraphs(list1,list2,VarName1,VarName2):
    ##This function will create four graphs.
    ## hspace is the space between top of graphs
    ## wspace is the space between graphs
    hspace=.5
    wspace=.4
    mpp.subplots_adjust(hspace=hspace, wspace=wspace )

    ##THE BAR CHART
    ## Create a bar plot of each var
    ## The values must first be catgorized
    ## Determine the shape of the bar graph

    mpp.subplot(321)
    xvals=[1,2,3,4,5]

    ##Create y values from the data
    rangeOfData=np.max(list1) - np.min(list1)
    ## Divide range into 5 categories and count
    # the number of data values in each category
    # Calculate a percentage in each category
    bar1=0
    bar2=0
    bar3=0
    bar4=0
    bar5=0
    counter=0
```

```
R5=rangeOfData/5

for j in list1:
    counter=counter+1
    if j < np.min(list1)+R5:
        bar1=bar1+1
    elif j >= np.min(list1)+R5 and j < np.min(list1)+(2*R5):
        bar2=bar2+1
    elif j >= np.min(list1)+(2*R5) and j < np.min(list1)+(3*R5):
        bar3=bar3+1
    elif j >= np.min(list1)+(3*R5) and j < np.min(list1)+(4*R5):
        bar4=bar4+1
    else:
        bar5=bar5+1

# Create names for the bars
B1="0-"+str(np.min(list1)+R5-1)
B2=str(np.min(list1)+R5) + "-" + str(np.min(list1)+(2*R5)-1)
B3=str(np.min(list1)+(2*R5)) + "-" + str(np.min(list1)+(3*R5)-1)
B4=str(np.min(list1)+(3*R5)) + "-" + str(np.min(list1)+(4*R5)-1)
B5=str(np.min(list1)+(4*R5)) + "-" + str(np.max(list1))

yvals1=[bar1,bar2,bar3,bar4,bar5]
barnames=[B1, B2, B3, B4, B5]
mpp.xticks(xvals, barnames, rotation=30)
mpp.bar(xvals, yvals1, facecolor='#9999ff', edgecolor='white', align='center')
name="Bar Graph for "+ VarName1 + "\n"
mpp.title(name)
mpp.show()
##---------------end BAR-----

## Create the Scatterplot
xvals=np.array(list1)
yvals=np.array(list2)

mpp.subplot(322)
mpp.scatter(xvals,yvals)
name="Scatter "+ VarName1 + " and " + VarName2 +"\n"
mpp.title(name)
mpp.show()
##----------end SCATTER----

## BOX PLOT for Each Variable

mpp.subplot(323)
mpp.boxplot(list1)
mpp.xlabel(VarName1)
name="Box Plot for "+ VarName1+"\n"
mpp.title(name)
mpp.show()

mpp.subplot(324)
mpp.boxplot(list2)
mpp.xlabel(VarName2)
name="Box Plot for "+ VarName2+"\n"
mpp.title(name)
mpp.show()
##---------end BOX--------

## Create a PIE graph of each var
## For pie or bar graphs, data must first be categorized
```

```
    #Get the average
    vavg=np.average(list1)
    v2avg=np.average(list2)

    #print(np.sort(list1))
    #The following will get Q1 and Q3 - but we do not need these
    #v25=np.percentile(list1,25)
    #v75=np.percentile(list1,75)

    y1=[]
    y2=[]

    for j in list1:
        #Those below average
        if 0<=j<vavg:
            y1.append(j)
        else:
            #Those above average
            y2.append(j)

    num_y1=len(y1)
    num_y2=len(y2)

    #Build the x values for the pie
    xvals=(num_y1,num_y2)
    labels="Above Avg", "Below Avg"
    #print(xvals)
    mpp.subplot(325)
    mpp.pie(xvals, labels=labels, autopct='%1.1f%%')
    name="Pie Graph for "+ VarName1 + "\n"
    mpp.title(name)
    mpp.show()


    ##--do this for list 2 as well
    y1=[]
    y2=[]

    for j in list2:
        #Those below average
        if 0<=j<v2avg:
            y1.append(j)
        else:
            #Those above average
            y2.append(j)

    num_y1=len(y1)
    num_y2=len(y2)

    #Build the x values for the pie
    xvals2=(num_y1,num_y2)
    labels="Below Avg", "Above Avg"
    #print(xvals)

    mpp.subplot(326)
    mpp.pie(xvals2,labels=labels,autopct='%1.1f%%')
    name="Pie Graph for "+ VarName2 + "\n"
    mpp.title(name)

    mpp.show()


#-----------------------------------------------------------------
```

```
def EvaluateData(UserInVar1,UserInVar2,VarNameDict,numlines,TheData):
    ###---This function creates and prints the summary statistics

    ## Create two blank lists
    var1_list=[]
    var2_list=[]

    #loop through all items in TheData to gather the vars of interest
    for i in range(numlines):
        if i > 0:
            Columns=TheData[i].rsplit("\t")
            next1=eval(Columns[VarNameDict[UserInVar1[0]]])
            next2=eval(Columns[VarNameDict[UserInVar2[0]]])
            #print(next1)
            var1_list.append(next1)
            var2_list.append(next2)

    ## Print out summary stats for both variables

    print("\nSUMMARY STATS FOR ", UserInVar1, ":")
    print("The mean for ", UserInVar1," is ", round(np.mean(var1_list)))
    print("The median for ", UserInVar1,"is ",round(np.median(var1_list)))
    print("The variance for ", UserInVar1, " is ",round(np.var(var1_list)))
    print("The standard deviation for ", UserInVar1, " is ", round(np.std(var1_list)))
    print("The max for ", UserInVar1, " is ", round(np.max(var1_list)))
    print("The min for ", UserInVar1, " is ", round(np.min(var1_list)))

    #print(var2_list)

    print("\nSUMMARY STATS FOR ", UserInVar2, ":")
    print("The mean for ", UserInVar2," is ", round(np.mean(var2_list)))
    print("The median for ", UserInVar2,"is ", round(np.median(var2_list)))
    print("The variance for ", UserInVar2, " is ", round(np.var(var2_list)))
    print("The standard deviation for ", UserInVar2, " is ", round(np.std(var2_list)))
    print("The max for ", UserInVar2, " is ", round(np.max(var2_list)))
    print("The min for ", UserInVar2, " is ", round(np.min(var2_list)))


    #return the two lists of the values for the two variables.
    return var1_list,var2_list


########Call main

main()
```

Example output for the above program:

```
Welcome

This program will allow you to choose any two variables and

it will print out basic summary statistics and graphs,

for the two variables selected.

_____________________________________________________________
```

```
The data set used for this program is called, Ch9DataSet.txt
The VARIABLES in the dataset are:  ID, CLA \(avg classgrade\)
LET \(for final class letter grade\), SEX, AGE,

GPA, HRS \(for hours of study per week\)
and FINAL \(for final exam score\)
_____________________________________________________________


Enter the name of the first numerical variable to include in the analysis: ID

Please select a numerical variable for analysis


Enter the name of the first numerical variable to include in the analysis: AGE

Enter the name of the second numerical variable to include in the analysis: Letter
Grade

Please select a numerical variable for analysis


Enter the name of the first numerical variable to include in the analysis: FINAL

SUMMARY STATS FOR  AGE :
The mean for  AGE  is  34.0
The median for  AGE is  33.0
The variance for  AGE  is  112.0
The standard deviation for  AGE  is  11.0
The max for  AGE  is  67
The min for  AGE  is  19

SUMMARY STATS FOR  FINAL :
The mean for  FINAL  is  84.0
The median for  FINAL is  86.0
The variance for  FINAL  is  145.0
The standard deviation for  FINAL  is  12.0
The max for  FINAL  is  99.0
The min for  FINAL  is 55.0
```