Python Graphics

A Reference for Creating 2D and 3D Images

B.J. Korites

Apress®

Python Graphics

A Reference for Creating 2D and 3D Images

B.J. Korites

Python Graphics

B.J. Korites
Duxbury, Massachusetts, USA

ISBN-13 (pbk): 978-1-4842-3377-1 ISBN-13 (electronic): 978-1-4842-3378-8
https://doi.org/10.1007/978-1-4842-3378-8

Library of Congress Control Number: 2018946635

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Todd Green
Development Editor: James Markham
Coordinating Editor: Jill Balzano

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484233771. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

For Pam

Table of Contents

About the Author

B.J. Korites has been involved in engineering and scientific applications of computers for his entire career. He has been an educator, consultant, and author of more than ten books on geometric modelling, computer graphics, artificial intelligence, simulation of physical processes, structural analysis, and the application of computers in science and engineering. He has been employed by Northrop Corporation, the Woods Hole Oceanographic Institute, Arthur D. Little, Itek, and Worcester Polytech. He has consulted for Stone and Webster Engineering, Gould Inc, Wyman Gordon, CTI Cryogenics, the US Navy, Aberdeen Proving Grounds, and others. Early in his career he developed mathematics and software that would find physical interferences between three-dimensional solid objects. This found wide application in the design of nuclear power plants, submarines, and other systems with densely packed spaces. He enjoys sailing and painting maritime landscapes in oils. He holds degrees from Tufts and Yale.

About the Technical Reviewer

Andrea Gavana has been programming in Python for almost 15 years and dabbling with other languages since the late nineties. He graduated from university with a Master's degree in Chemical Engineering, and he is now a Senior Reservoir Engineer working for Maersk Oil in Copenhagen, Denmark.

Andrea enjoys programming at work and for fun, and he has been involved in multiple open source projects, all Python-based. One of his favorite hobbies is Python coding, but he is also fond of cycling, swimming, and cozy dinners with family and friends.

Acknowledgments

I would like to thank my wife, Pam, for her patience during the many long days and nights that I spent writing this book and for her understanding of the distant stare I sometimes had while off in another world thinking of math and Python, two of life's great joys. I would also like to thank everyone at Apress, especially editors Todd Green and Jill Balzano, who made the production of this book a fast and seamless process.

CHAPTER 1

Essential Python Commands and Functions

In this chapter, you will learn the essential Python commands and functions you will need to produce the illustrations shown in this book. You will learn how to use Python's basic plotting functions, set up a plotting area, create a set of two-dimensional coordinate axes, and use basic plotting primitives (the dot, the line, and the arrow), which are the building blocks you will use to construct images throughout this book. In Chapter 2, you will learn how to use these primitives to build two-dimensional images and then translate and rotate them. In Chapter 3, you will extend these concepts to three dimensions. Also in this chapter you will learn about colors, how to apply text to your plots, including the use of Latex commands, and the use of lists and arrays. By the last chapter, you will be able to create images such as Figure 1-1.

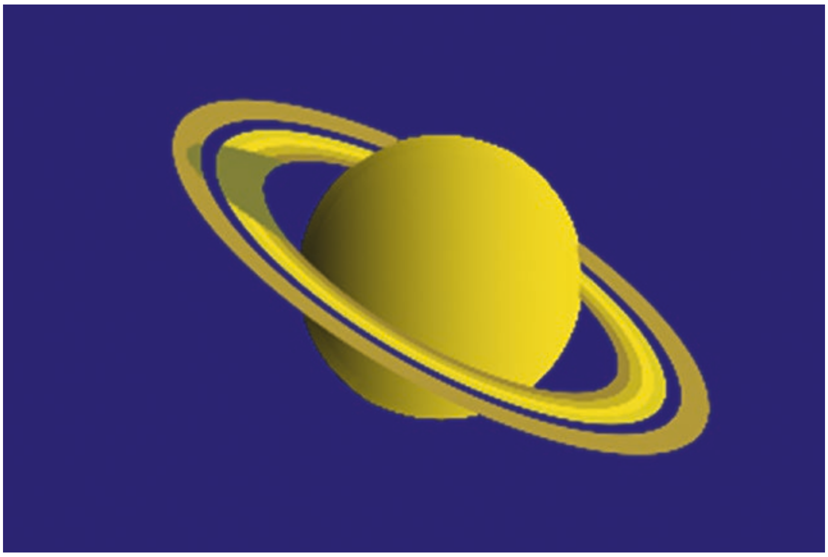

Figure 1-1. *Saturn*

B.J. Korites, *Python Graphics*, https://doi.org/10.1007/978-1-4842-3378-8_1

1.1 Programming Style

First a note on the programming style used in this book. We all have preferences when it comes to style. I favor a clear, top-down, open style. Many programmers try to reduce their code to as few lines as possible. That may be fine in practice but in an instructional text, such as we have here, I believe it is better to proceed slowly in small, simple steps. The intention is to keep everything clear and understandable. Since I do not know the skill level of the readers, and since I want to make this book accessible to as wide an audience as possible, I generally start each topic from an elementary level, although I do assume some familiarity with the Python language. If you are just learning Python, you will benefit from the material in this first chapter. If you are an accomplished Pythoner, you could probably skip it and move on to Chapter 2.

Some Python developers advocate using long descriptive names for variables such as "temperature" rather than "T." I find excessively long variable names make the code difficult to read. It's a matter of preference. With relatively short programs such as we have in this book, there's no need for complex programming. Try to adopt a style that is robust rather than elegant but fragile.

My programs usually have the same structure. The first few statements are generally **import numpy as np**, **import matplotlib.pyplot as plt**, and so on. Sometime I will import from the **math** library with **from math import sin, cos, radians, sqrt**. These are commonly used functions in graphics programming. Importing them separately in this way eliminates the need to use prefixes as in **np.sin();** you can just use **sin()**. Then I most often define the plotting area with **plt.axis([0,150,100,0])**. As explained in Section 1.2, these values, where the x axis is 50% wider than the y axis, produce a round circle and a square square without reducing the size of the plotting area. At this point, axes can be labelled and the plot titled if desired. Next, I usually define parameters (such as diameters, time constants, and so on) and lists. Then I define functions. Finally, in lengthy programs, at the bottom I put a control section that invokes the functions in the proper order.

Including **plt.axis('on')** plots the axes; **plt.grid(True)** plots a grid. They are very convenient options when developing graphics. However, if I do not want the axes or grid to show in the final output, I replace these commands with **plt.axis('off')** and **plt.grid(False)**. The syntax must be as shown here. See Section 1.10 to learn how to create your own grid lines if you are not satisfied with Python's defaults.

I often begin development of graphics by using the **scatter()** function which produces what I call *scatter dots*. They are fast and easy to use and are very useful in the development stage. If kept small enough and spaced closely together, dots can produce acceptable lines and curves. However, they can sometimes appear a bit fuzzy so, after I have everything working right, I will often go back and replace the dots with short line segments using either arrows via **plt.arrow()** or lines via **plt.plot()**. There is another aspect to the choice of dots or lines: which overplots which. You don't want to create something with dots and then find lines covering it up. This is discussed in Section 1.14.

Some variants of Python require the **plt.show()** statement at the end to plot graphics. My setup, Anaconda with Spyder and Python 3.5 (see Appendix A for installation instructions), does not require this but I include it anyway since it serves as a marker for the end of the program. Finally, press the F5 key or click on the Run button at the top to see what you have created. After you are satisfied, you can save the plot by right-clicking it and specifying a location.

Regarding the use of lists, tuples and arrays, they can be a great help, particularly when doing graphics programming that involves a lot of data points. They are explained in Section 1.19.5. An understanding of them, together with a few basic graphics commands and techniques covered in this chapter, are all you need to create the illustrations and images you see in this book.

1.2 The Plotting Area

A computer display with a two-dimensional coordinate system is shown in Figure 1-2. In this example, the origin of the x,y coordinate axes, (x=0, y=0), is located in the center of the screen. The positive x axis runs from the origin to the right; the y axis runs from the origin vertically downward. As you will see shortly, the location of the origin can be changed as can the directions of the x and y axes. Also shown is a point p at coordinates (x,y), which are in relation to the x and y axes.

The direction of the y axis pointing down in Figure 1-2 may seem a bit unusual. When plotting data or functions such as y=cos(x) or y=exp(x), we usually think of y as pointing up. But when doing technical graphics, especially in three dimensions, as you will see later, it is more intuitive to have the y axis point down. This is also consistent with older variants of BASIC where the x axis ran along the top of the screen from left to right and the y axis ran down the left side. As you will see, you can define y to point up or down, whichever best suits what you are plotting.

1.3 Establishing the Size of the Plotting Area

The plotting area contains the graphic image. It always appears the same *physical* size when displayed in the Spyder output pane. Spyder is the programming environment (see Appendix A). However, the *numerical* size of the plotting area, and of the values of the point, line, and arrow definitions within the plotting area, can be specified to be anything. Before doing any plotting, you must first establish the area's *numerical* size. You must also specify the location of the coordinate system's origin and the directions of the coordinate axes. As an illustration, Listing 1-1 uses the **plt.axis([x1,x2,y1,y2])** function in line 8 to set up an area running from x=-10 to +10; y=—10 to +10. The rest of the script will be explained shortly.

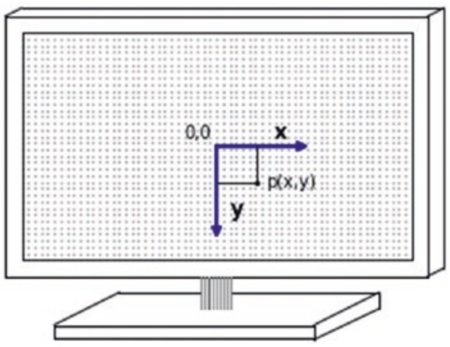

Figure 1-2. *A two-dimensional x,y coordinate system with its origin (0,0) centered in the screen. A point p is shown at coordinates (x,y) relative to x,y.*

Listing 1-1. Program PLOTTING_AREA

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x1=-10
5  x2=10
6  y1=-10
7  y2=10
8  plt.axis([x1,x2,y1,y2])
9
10 plt.axis('on')
```

```
11 plt.grid(True)
12
13 plt.show()
```

Listing 1-1 produces the plotting area shown in Figure 1-3. It has a horizontal width of 20 and a vertical height of 20. I could have made these numbers 200 and 200, and the area would appear in an output pane as the same physical size but with different numerical values on the axes. Line 13 contains the command **plt.show()**. The purpose of this is to display the program's results in the output pane. With modern versions of Python it isn't required since the plots are automatically displayed when the program is run. With older versions it may or may not be displayed. **plt.show()** can also be placed within a program in order to show plots created during execution. Even though it may not be necessary, it's a good idea to include this command at the end of your script since it can serve as a convenient marker for the end of your program. Lines 1, 2, 10, and 11 in Listing 1-1 will be explained in the following sections. These commands, or variations of them, will appear in all of our programs.

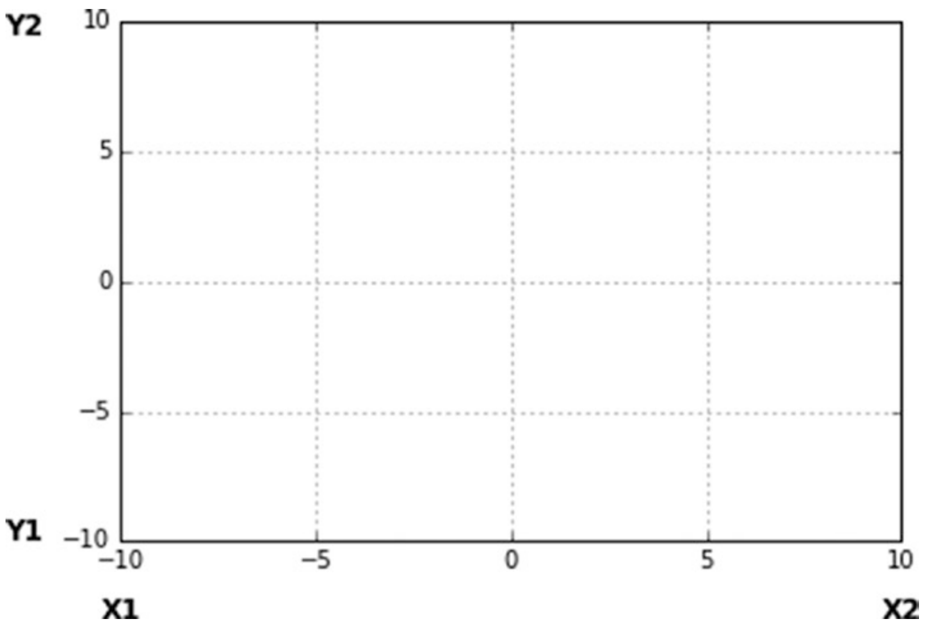

Figure 1-3. *Plotting area produced by Listing 1-1 with (0,0) located in the center of the area*

1.4 Importing Plotting Commands

While Python has many built-in commands and functions available, some math and graphics commands must be imported. Lines 1 and 2 in Listing 1-1 do this. The **import numpy as np** statement in line 1 imports math functions such as **sin(ϕ)**, **e**$^{\alpha}$, and so on. The **np** in this statement is an abbreviation that may be used when referring to a **numpy** function. When used in a program, these functions must be identified as coming from **numpy**. For example, if you were to code **v=e**$^{\alpha}$, the program statement would be **v=np.exp(α)** where **α** would have been previously defined. You don't have to write out the full length **numpy.exp(α)** since you defined the shorthand **np** for **numpy** in line 1. Graphics commands are handled similarly. The statement **import matplotlib.pyplot as plt** imports the library **pyplot,** which contains graphics commands. **plt** is an abbreviation for **pyplot**. For example, if you want to plot a dot at x,y you would write **plt.scatter(x,y)**. I will talk more about **plt.scatter()** shortly.

Functions may also be imported directly from **numpy**. The statement **from numpy import sin, cos, radians** imports the **sin()**, **cos()**, and **radians()** functions. When imported in this manner they may be used without the **np** prefix. There is also a **math** library that operates in a similar way. For example, **from math import sin, cos, radians** is equivalent to importing from **numpy**. You will be using all these variations in the coming programs.

There is also a graphics library called **glib** that contains graphics commands. **glib** uses a different syntax than **pyplot**. Since **pyplot** is used more widely, you will use it in your work here.

Line 8 in Listing 1-1, **plt.axis([x1,x2,y1,y2])**, is the standard form of the command that sets up the plotting area. This is from the **pyplot** library and is preceded by the **plt.** prefix. There are attributes to this command and there are other ways of defining a plotting area, notably the **linspace()** command, but the form in line 8 is sufficient for most purposes and is the one you will use. x1 and x2 define the values of the left and right sides, respectively, of the plotting area; y1 and y2 define the bottom and top, respectively. With the numeric values in lines 8-11 you get the plotting area shown in Figure 1-3. x1,x2,y1, and y2 always have the locations shown in Figure 1-3. That is, x1 and y1 always refer to the lower left corner, y2 to other end of the y axis, and x2 to the other end of the x axis. Their values can change, but they always refer to these locations. They may be negative, as shown in Figure 1-4.

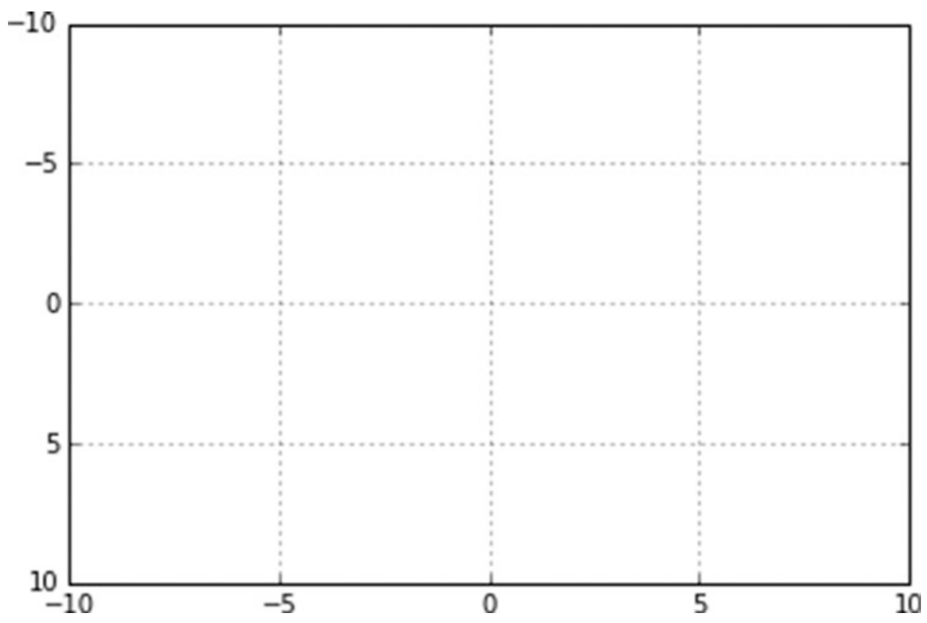

***Figure 1-4.** Plotting area with (0,0) located in the center, positive y direction pointing down*

Because the x and y values specified in lines 4-7 are symmetric in both the x and y directions (i.e. −10, +10), this plotting area has the (x=0, y=0) point halfway between. In this case, the center of the area will be the origin used as reference for plotting coordinates. Since x1 < x2, the positive direction of the x axis will run horizontally from left to right. Similarly, since y1 < y2, the positive direction of the y axis will go vertically up. But earlier I said we want the positive y direction to go vertically down. You can do that by reversing the y values to y1=10, y2=−10. In this case, you get the area shown in Figure 1-4 where the positive x axis still goes from left to right but the positive y axis now points down. The center is still in the middle of the plotting area.

You could move the origin of the coordinate system off center by manipulating x1, x2,y1, and y2. For example, to move the x=0 point all the way to the left side, you could specify x1=0, x2=20. To move the (x=0, y=0) point to the lower left corner, you could specify x1=0, x2=20, y1=0, y2=20. But that would make the positive y direction point up; you want it to point down, which you can do by making y2=0, y1=20. This will make the origin appear in the *upper* left corner. You are free to position the (0,0) point anywhere, change the direction of positive x and y, and scale the numerical values of the coordinate axes to suit the image you will be trying to create. The numerical values you are using here could be anything. The physical size of the plot produced by Python will be the same; only the values of the image coordinates will change.

1.5 Displaying the Plotting Area

In line 10 of Listing 1-1 the statement **plt.axis('on')** displays the plotting area with its frame and numerical values. If you omit this command, the frame will still be displayed with numerical values. So why include this command? Because, when creating a plot it is sometimes desirable to turn the frame off. To do that, replace **plt.axis('on')** with **plt.axis('off')**. Having the command there ahead of time makes it easy to type **'off'** over **'on'** and vice versa to switch between the frame showing and not showing. Also, after you have finished with a plot, you may wish to use it in a document, in which case you may not want the frame. Note that **'on'** and **'off'** must appear in quotes, either single or double.

1.6 The Plotting Grid

Line 11 of Listing 1-1, **plt.grid(True),** turns on the dotted grid lines, which can be an aid when constructing a plot, especially when it comes time to position textual information. If you do not include this command, the grid lines will not be shown. To turn off the grid lines, change the **True** to **False**. Note the first letter in **True** and **False** is capitalized. **True** and **False** do *not* appear in quotations marks. As with **plt.axis()**, having the **plt.grid(True)** and **plt.grid(False)** commands there makes it easy to switch back and forth. Again, note that both **True** and **False** must have the first letter capitalized and do *not* appear in quotes.

1.7 Saving a Plot

The easiest way to save a plot that appears in the output pane is to put your cursor over it and right-click. A window will appear allowing you to give it a name and specify where it is to be saved. It will be saved the .png format. If you are planning to use it in a program such as Photoshop, the .png format works. Some word processing and document programs may require the .eps (encapsulated Postscript) format. If so, save it in the .png format, open it in Photoshop, and resave it in the .eps format. It’s a bit cumbersome but it works.

1.8 Grid Color

There are some options to the **plt.grid()** command. You can change the color of the grid lines with the **color='color'** attribute. For example, **plt.grid(True, color='b')** plots a blue grid. More color options will be defined shortly.

1.9 Tick Marks

The **plt.grid(True)** command will create a grid with Python's own choice of spacing, which may not be convenient. You can alter the spacings with the **plt.xticks(xmin, xmax, dx)** and **plt.yticks(ymin, ymax, dy)** commands. **min** and **max** are the range of the ticks; **dx** and **dy** are the spacing. While normally you want the tick marks to appear over the full range of x and y, you can have them appear over a smaller range if you wish. These commands appear in lines 23 and 24 of Listing 1-2.

Listing 1-2. Program TICK_MARKS

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  #————————————————plotting area
5  x1=-10
6  x2=140
7  y1=90
8  y2=-10
9  plt.axis([x1,x2,y1,y2])
10 plt.axis('on')
11
12 #——————————————————grid
13 plt.grid(True,color='b')
14 plt.title('Tick Mark Sample')
15
16 #————————————————tick marks
```

```
17 xmin=x1
18 xmax=x2
19 dx=10
20 ymin=y1
21 ymax=y2
22 dy=-5
23 plt.xticks(np.arange(xmin, xmax, dx))
24 plt.yticks(np.arange(ymin, ymax, dy))
25
26 plt.show()
```

The output is shown in Figure 1-5. In line 23, **xmin** and **xmax** are the beginning and end of the range of ticks along the x axis, similarly for line 24, which controls the y axis ticks. **dx** in line 19 spaces the marks 10 units apart from x1=-10 (line 5) to x2=140 (line 6). **dy** in line 22 is -5. It is negative because y2=−10 (line 8) while y1=+90 (line 7). Thus, as the program proceeds from y1 to y2, y decreases in value; hence **dy** must be negative.

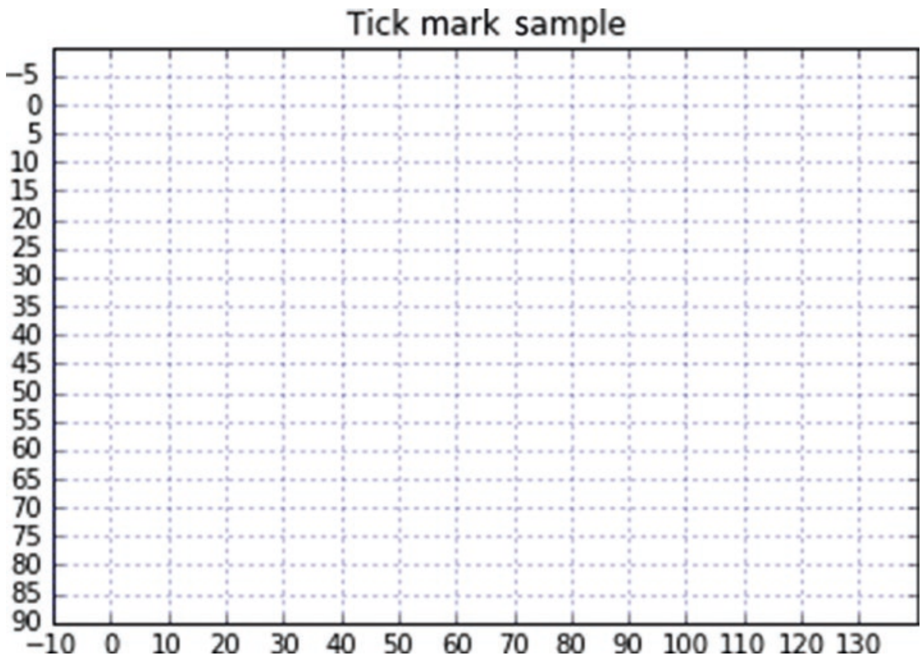

Figure 1-5. *User-defined tick mark*

1.10 Custom Grid Lines

The automatically generated grid that is produced by the **plt.grid(True)** command is not always satisfactory especially if you want to include text in your plot. It is often not fine enough to accurately place text elements. But if the **xtick()** and **ytick()** commands are used to reduce the spacing, the numbers along the axes can become cluttered. The numbers can be eliminated but then you do not have the benefit of using them to position textual information such as when labelling items on a plot. The grid shown in Figure 1-3 would be more helpful if the increments were smaller. You can produce your own grid lines and control them any way you want. The code in Listing 1-3 produces Figure 1-6, a plotting area with finer spacing between grid lines.

Listing 1-3. Program CUSTOM_GRID

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

x1=-5
x2=15
y1=-15
y2=5
plt.axis([x1,x2,y1,y2])

plt.axis('on')

dx=.5                                     #x spacing
dy=.5                                     #y spacing
for x in np.arange(x1,x2,dx):             #x locations
     for y in np.arange(y1,y2,dy):        #y locations
     plt.scatter(x,y,s=1,color='grey')    #plot a grey point at x,y

plt.show()
```

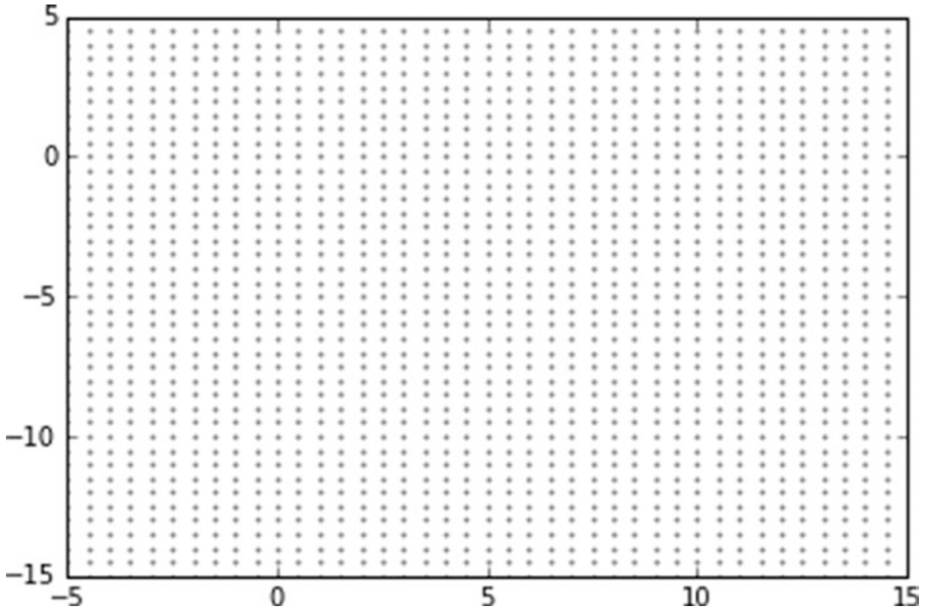

Figure 1-6. *Plotting area with custom grid*

The **scatter()** function in line 16 of Listing 1-3 plots a grey dot at every x,y location. I will discuss **scatter()** in more depth later. Note that **plt.grid(True)** is not used in this program. Lines 1-10 produce the plotting area with axes as before. This time, instead of using the **plt.grid(True)** command, you produce your own custom grid in lines 12-16. Lines 12 and 13 specify the spacing. The loop beginning at line 14 advances horizontally from left to right in steps **dx**. The loop beginning at line 15 does the same in the vertical direction. The size of the dot is specified as 1 by the **s=1** attribute in line 16. This could be changed: **s=.5** will give a smaller dot; **s=5** will give a larger one. The **color='grey'** attribute sets the dot color to grey. You can experiment with different size dots, colors, and spacings. Sometimes it can be beneficial to use the grid produced by **Grid(True)** along with a custom grid.

1.11 Labelling the Axes

Axes can be labelled with the **plt.xlabel('label')** and **plt.ylabel('label')** functions. As an example, the lines

```
plt.xlabel('this is the x axis')
plt.ylabel('this is the y axis')
```

when added to Listing 1-3 after line 10 produce Figure 1-7 where the custom grid dots have been changed to a lighter grey by using the attribute **color='lightgrey'** in the **plt.scatter()** function.

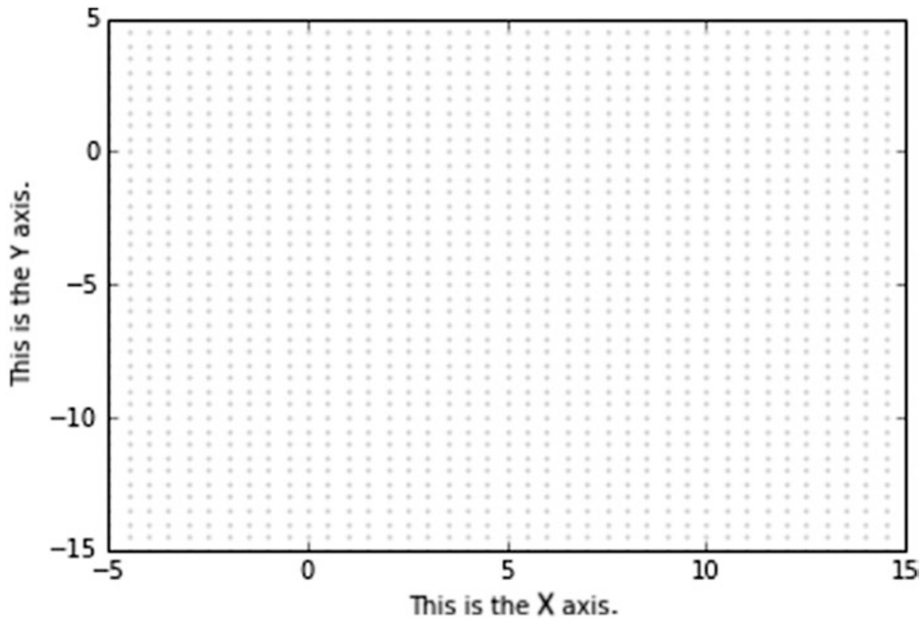

Figure 1-7. *Plotting area with axis labels and custom grid*

In Figure 1-8 you can see the matplotlib grid. This combination of Python's grid plus a custom grid makes a convenient working surface for locating elements.

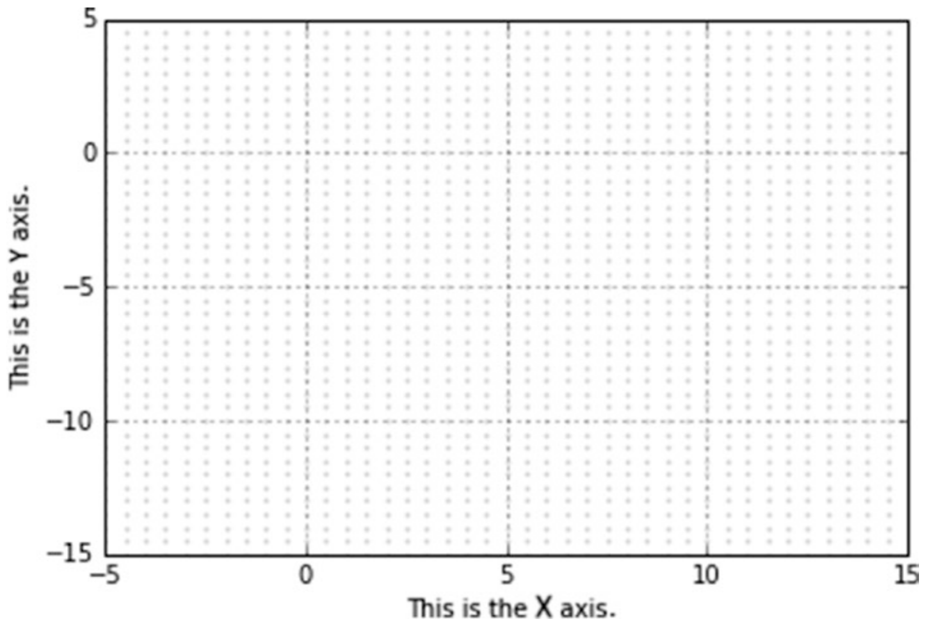

Figure 1-8. *Plotting area with axis labels, the Python grid, and a custom grid*

1.12 The Plot Title

Your plot can be titled easily with the **plt.title('title')** statement. Inserting the following line produces Figure 1-9:

```
plt.title('this is my plot')
```

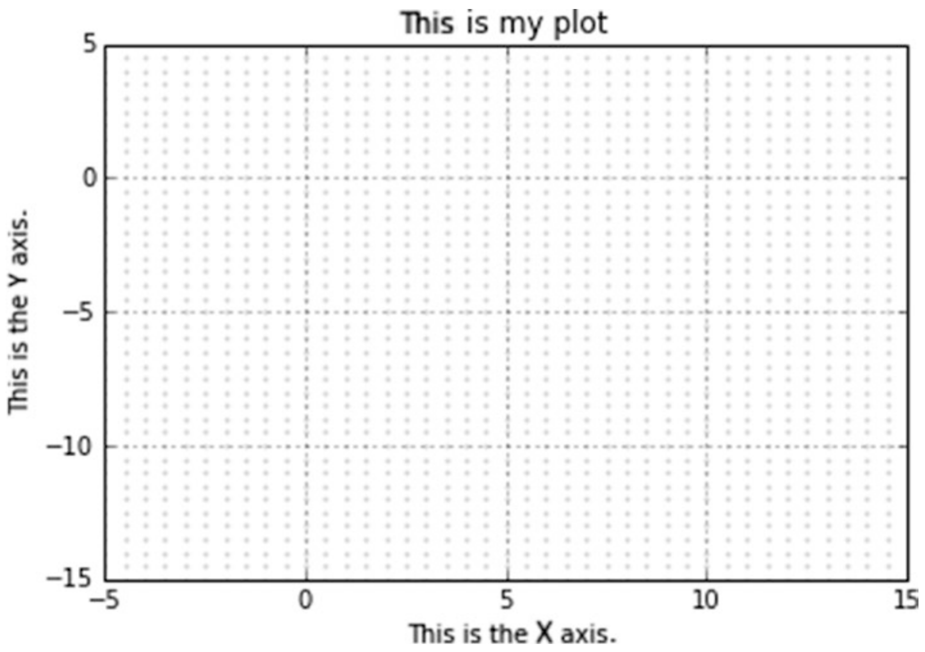

Figure 1-9. *Plotting area with axis labels, Python grid, custom grid, and title*

1.13 Colors

As you move along in this book, you will make good use of Python's ability to plot in color. Some of the colors available are

'k' for black

'b' for blue

'c' for cyan

'g' for green

'm' for magenta

'r' for red

'y' for yellow

'gray' or 'grey'

'lightgray' or 'lightgrey'

For example, the following statement will plot a green dot at coordinates x,y:

```
plt.scatter(x,y,color='g')
```

A swatch of many more colors can be found at

https://matplotlib.org/examples/color/named_colors.html.

The color attribute may be used in the **scatter()**, **plot()**, and **arrow()** functions along with other attributes.

1.13.1 Color Mixing

You can mix your own hues from the primary colors of red (r), green (g), and blue (b) with the specification **color=(r,g,b)** where r,g,b are the values of red, green, and blue in the mix, with values of each ranging from 0 to 1. For example **color=(1,0,0)** gives pure red; **color=(1,0,1)** gives magenta, a purplish mix of red and blue; **color=(0,1,0)** gives green; **color(.5,0.1)** gives more red and less blue in the magenta; **color(0,0,0)** gives black; and **color(1,1,1)** gives white. Keeping the r,g,b values the same gives a grey progressing from black to white as the values increase. That is, **color=(.1,.1,.1)** produces a dark grey, **color(.7,.7,.7)** gives a lighter grey, and **color(.5,.9,.5)** gives a greenish grey. Note that when specifying **'grey'** it can also be spelled **'gray'**.

Listing 1-4 shows how to mix colors in a program. Lines 7-9 establish the fraction of each color ranging from 0-1. The red component in line 7 depends on x, which ranges from 1-100. The green and blue components each have a value of 0 in this mix. Line 10 draws a vertical line at x from top to bottom having the color mix specified by the attribute **color=(r,g,b)**. The results are shown in Figure 1-10. The hue on the left side is almost black. This is because the amount of each color in the mix is 0 or close to it (**r=.01,g=0,b=0**). The hue on the right is pure red since on that side **r=1,g=0,b=0**; that is, the red is full strength and is not contaminated by green or blue.

Listing 1-4. Program COLORS

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
```

```
plt.axis([0,100,0,10])

for x in np.arange(1,100,1):
    r=x/100
    g=0
    b=0
    plt.plot([x,x],[0,10],linewidth=5,color=(r,g,b))

plt.show()
```

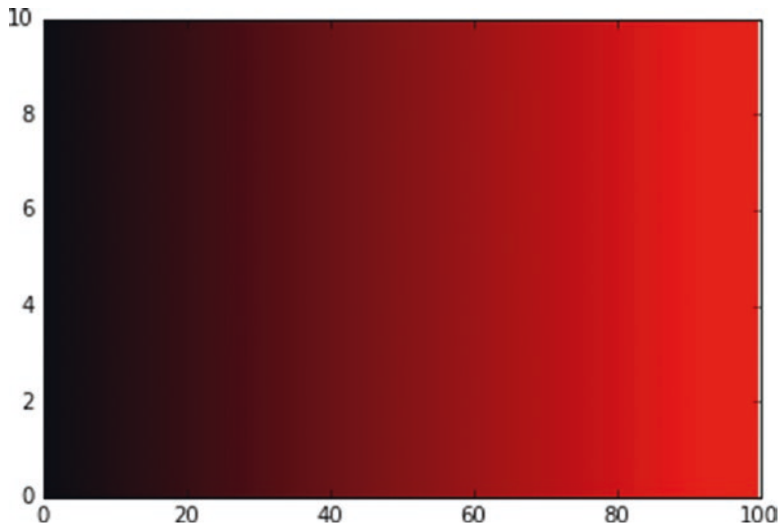

Figure 1-10. *Red color band produced by Listing 1-4 with r=x/100, g=0, b=0*

Figure 1-11 shows the result of adding blue to the mix. Figure 1-12 shows the result of adding green to the red. Mixing all three primary colors equally gives shades of grey ranging from black to white, as shown in Figure 1-13.

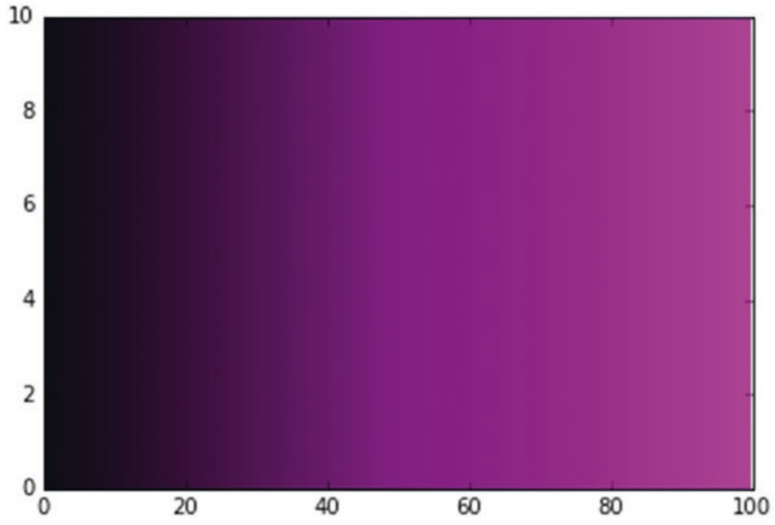

Figure 1-11. *Purple color band with r=x/100, g=0, b=x/100*

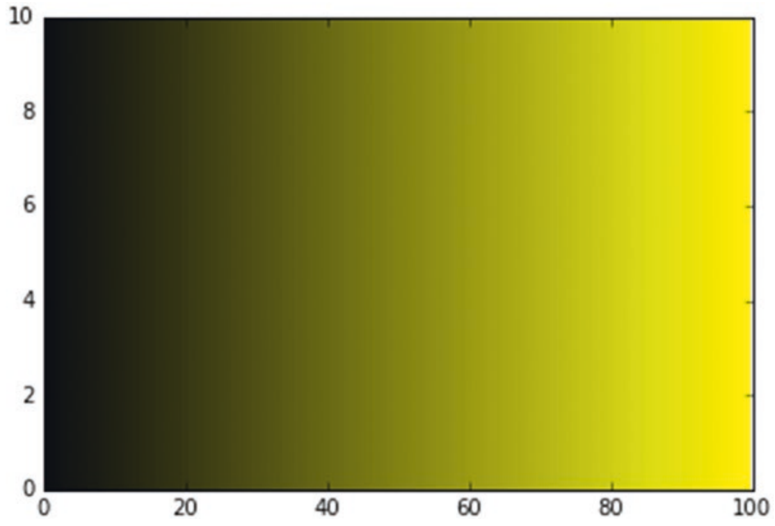

Figure 1-12. *Yellow color band with r=x/100, g=x/100, b=0*

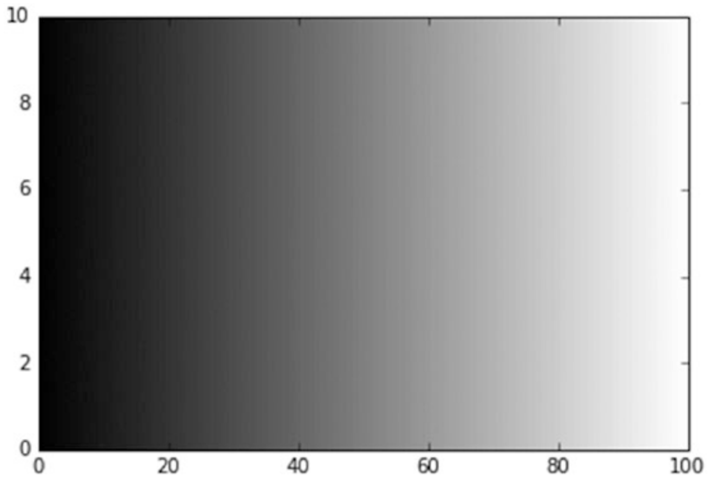

Figure 1-13. *Grey color band with r=x/100, g=x/100, b=x/100*

There are 256 values of each primary color available. Mixing them, as I did here, gives 256^3, which is almost 17 million different hues.

1.13.2 Color Intensity

The intensity of a color can be controlled with the **alpha** attribute, as shown in lines 6-8 in Listing 1-5, which produced Figure 1-14. **alpha** can vary from 0 to 1, with 1 producing the strongest hue and 0 the weakest.

Listing 1-5. Program COLOR_INTENSITY

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  plt.axis([0,100,0,10])
5
6  plt.scatter(60,50,s=1000,color='b',alpha=1)
7  plt.scatter(80,50,s=1000,color='b',alpha=.5)
8  plt.scatter(100,50,s=1000,color='b',alpha=.1)
9
10 plt.show()
```