Chapter V

Introduction to Turtle Graphics

Chapter V Topics

5.1 Introduction

Python programming is about to become much more fun and interesting because we are now starting to learn one of my most favorite computer science concepts, *Graphics*. In this chapter, we will start with *Turtle Graphics*. This is a simplistic type of graphics that was actually introduced in the language Logo. Remember that Logo was designed to teach programming concepts to young children, so it and the *Turtle Graphics* it later introduced, are both fairly simplistic.

The whole idea behind *Turtle Graphics* is based on an imaginary robotic *turtle* with a pen tied to its tail. This turtle can be given simple commands like **forward** or **turn**. As the turtle moves, it draws a line on the ground showing the path it just traversed.

5.2 Importing Libraries and Turtle Graphics Setup

Many programming languages require you to "import libraries" on certain occasions. For example, in Python, if we want to use Turtle Graphics, we need to import the **turtle** library. For other specialized features, we need to import other libraries. What happens if we do not import the **turtle** library? Well, we will not have access to any of the Turtle Graphics commands. You may wonder why importing is even necessary. Why not simply make all of the commands from all of the libraries accessible all of the time? The answer is that some languages are huge and that would take up way too much space.

What most languages, like Python, do is there are some commands that are available without importing anything. These are commands like **print** or **type**. There are other commands that are organized in different libraries. To use them, you need to import that specific library at the beginning of your program. Since you only import the libraries that you need, the program does not waste storage space.

We will start things off with program **TurtleGraphics01.py**, shown in Figure 5.1. This program seems rather lengthy, even though it only has 5 lines of real code. The rest of the program consists of comments explaining the purpose of each of those 5 lines of code.

Figure 5.1

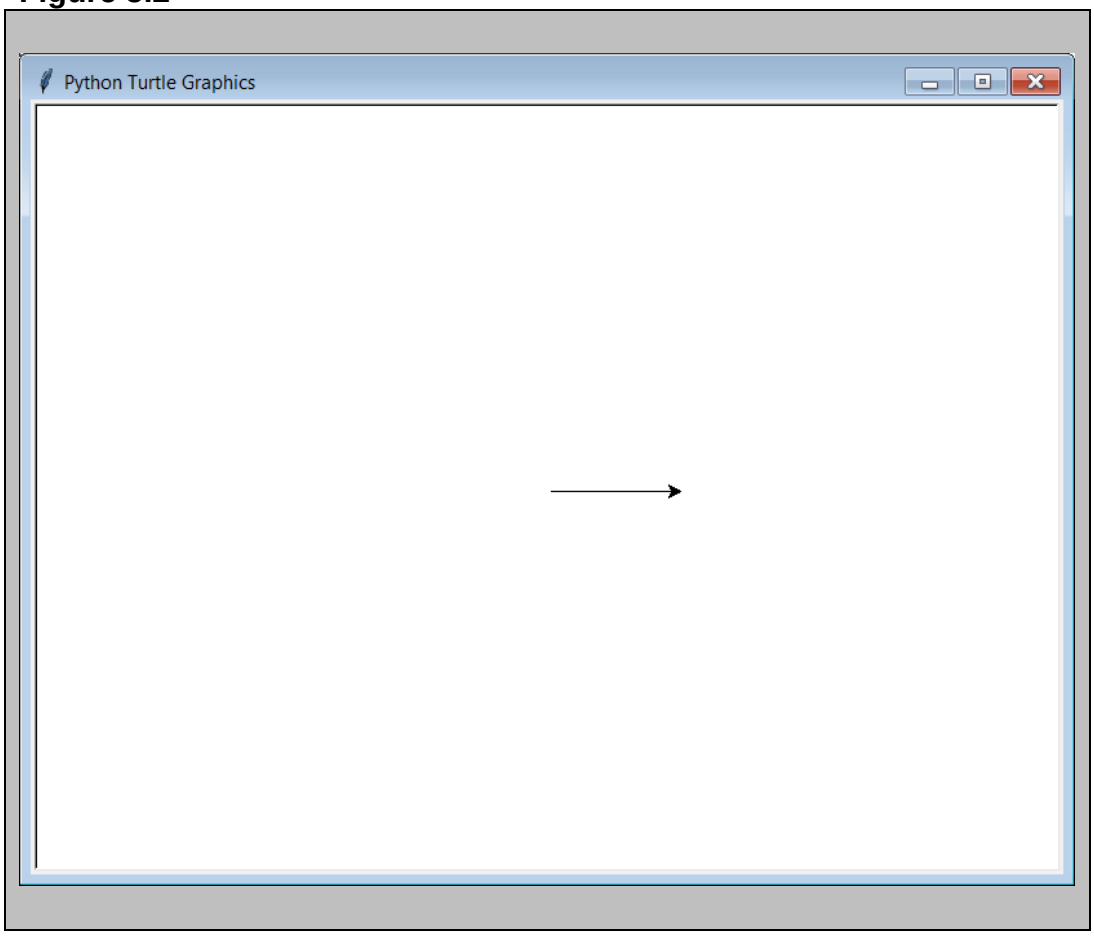
```
 1 # TurtleGraphics01.py
 2 # This program introduces "Turtle Graphics" by
 3 # importing the <turtle> library and drawing a
 4 # single line with the <forward> command.
 5
 6
 7 # Required to have access to
 8 # the turtle graphics commands.
 9 import turtle
10
11 # Specifies the dimensions of
12 # the Turtle Graphics window.
13 turtle.setup(800,600)
14
15 # Moves the "turtle" forward 100 pixels
16 # and draws a line in the process.
17 turtle.forward(100)
18
19 # Required to "update" the window
20 # when everything is drawn.
21 turtle.update()
22
23 # Required to keep the graphics window
24 # open when the program is finished.
25 turtle.done()
```

The program starts with an **import** command which gives us access to the **turtle** library and all of its commands. One of these commands is **setup** which is used to specify the dimensions of the Turtle Graphics window. Note the *library.command* syntax used for this and all of the remaining commands in this program. Next is the only command that actually draws anything. **turtle.forward(100)** makes the turtle move "forward" **100** pixels and draws a line in the process. The program finishes with an **update** command followed by a **done** command. **update** is used to "update" the window when you are finished drawing everything. **done** is used to keep the Turtle Graphics window open until the user closes it. Most of your Turtle Graphics programs will end with these 2

commands. For that matter, most of your Turtle Graphics programs will also begin with **import** followed by **setup**. It is all of the commands that you put in-between that makes the output interesting.

By default, the turtle starts in the center of the window and faces right. This may help explain the output shown in Figure 5.2. Keep in mind that for any graphics program, the output will not be in the **Output/Messages** window at the bottom of the window. A new window, like the one shown in Figure 5.2, will open.

Figure 5.2



The next couple programs will essentially do the same thing as the first program and have the same output. The difference will be in how the **turtle** library is imported. In program **TurtleGraphics02.py**, shown in Figure 5.3, the keyword **from** is added to the **import** statement, or rather **import** statements. In each of these I am telling the computer which command I want imported "from" the **turtle** library. One big benefit of using **from** with **import** is that the rest of your program can be less "wordy". Note how we can now say **setup(800,600)** instead of **turtle.setup(800,600)** or **forward(100)** instead of **turtle.forward(100)**. Essentially, when you use **from,** you no longer need to use the *library.command* syntax. You can just use the *command* and not worry about the *library*.

Figure 5.3

```
 1 # TurtleGraphics02.py
 2 # This program has the exact same output as the
 3 # previous one.  By importing the individual
 4 # <turtle> library command in this way, the
 5 # rest of the code can be less "wordy".
 6
 7
 8 from turtle import setup
 9 from turtle import forward
10 from turtle import update
11 from turtle import done
12
13 setup(800,600)
14 forward(100)
15 update()
16 done()
```

Program **TurtleGraphics03.py**, in Figure 5.4, makes things even less "wordy" by using a *wildcard* (the asterisk *) to combine all of the **import** commands from the previous program into one single command. Now, in one statement, we are importing all of the commands **from** the **turtle** library. I will use this **import** technique for the remaining program examples in this textbook.

Figure 5.4

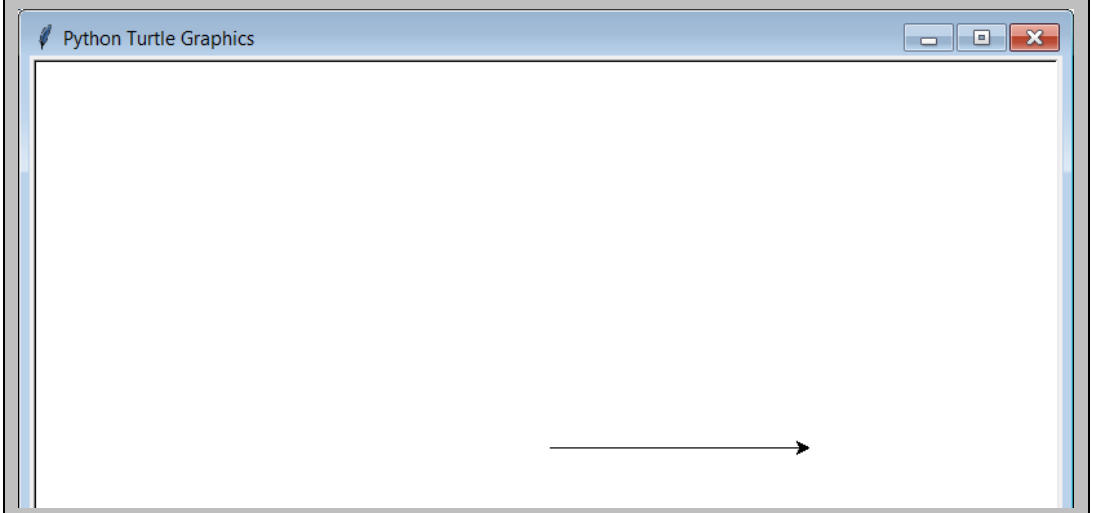
```
 1 # TurtleGraphics03.py
 2 # This program has the exact same output as the
 3 # previous two.  By using the <*> "wildcard" we
 4 # can <import> all of the <turtle> library commands
 5 # at once making the code even less "wordy".
 6
 7
 8 from turtle import *
 9
10 setup(800,600)
11 forward(100)
12 update()
13 done()
```

5.3 Drawing by Moving and Turning the Turtle

Now that we can **import** the **turtle** library and properly **setup** the Turtle Graphics window, we can start drawing things. Program **TurtleGraphics04.py**, in Figure 5.5, is almost identical to the previous program. The difference is the **forward(100)** command is called twice. This means that after moving "forward" **100** pixels, it move again another **100** pixels in the same direction. You would get the same result with a single **forward(200)** command. Either way, the result is a line twice as long as those generated by the previous 3 programs.

Figure 5.5

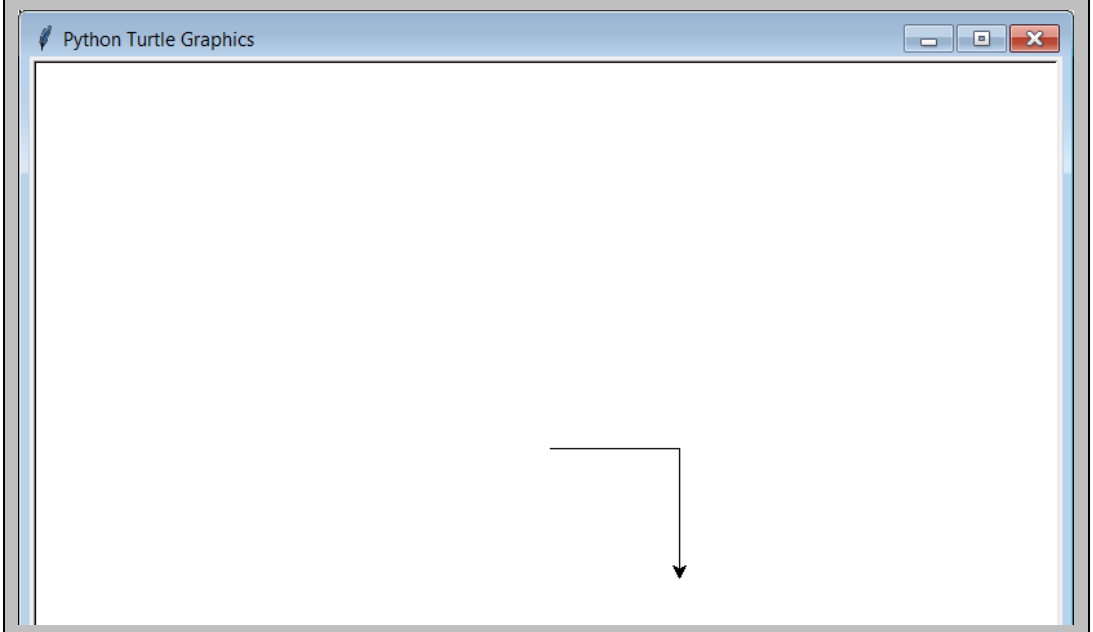
```
1  # TurtleGraphics04.py
2  # This program shows what happens when
3  # <forward> is called twice.
4
5
6  from turtle import *
7
8  setup(800,600)
9
10 forward(100)
11 forward(100)
12
13 update()
14 done()
```



Program **TurtleGraphics05.py**, in Figure 5.6, inserts a **right(90)** command in-between the 2 **forward(100)** commands. The result is that after the computer moves "forward" **100** pixels the first time, it makes a **90** degree turn to the "right" before it moves again.

Figure 5.6

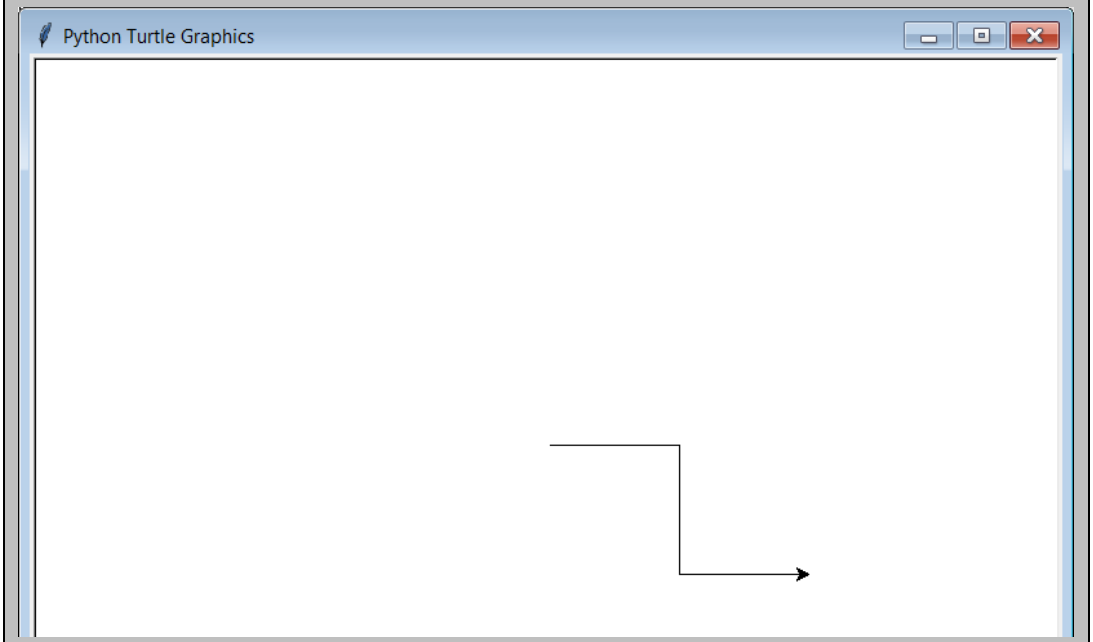
```
 1 # TurtleGraphics05.py
 2 # This program has the "turtle" make a
 3 # 90 degree turn to the "right" before
 4 # the second line is drawn.
 5
 6
 7 from turtle import *
 8
 9 setup(800,600)
10
11 forward(100)
12 right(90)
13 forward(100)
14
15 update()
16 done()
```



Program **TurtleGraphics06.py**, in Figure 5.7, demonstrates that the "turtle" can turn **left** as well.

Figure 5.7

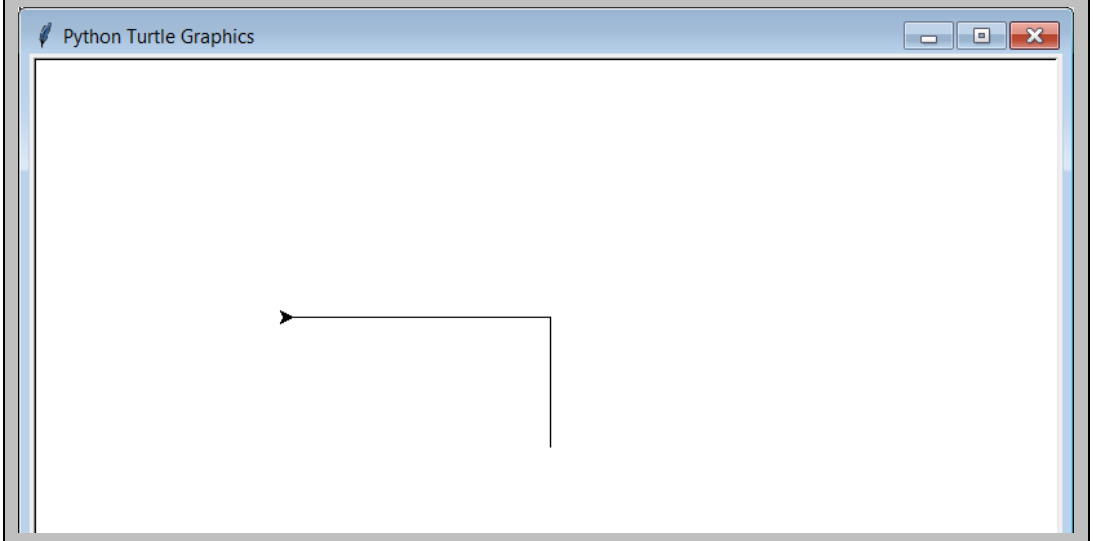
```
# TurtleGraphics06.py
# This program shows that "left" turns
# are also possible.


from turtle import *

setup(800,600)

forward(100)
right(90)
forward(100)
left(90)
forward(100)

update()
done()

```



Program **TurtleGraphics07.py**, in Figure 5.8, demonstrates a couple things. First, the "turtle" can also move **backward** as well as **forward**. You need to see the execution in on your computer to truly appreciate what is happening here. Second, it shows that the turtle does not need to start by moving to the right. You can turn before you move, which means you can start by moving in any direction.

Figure 5.8

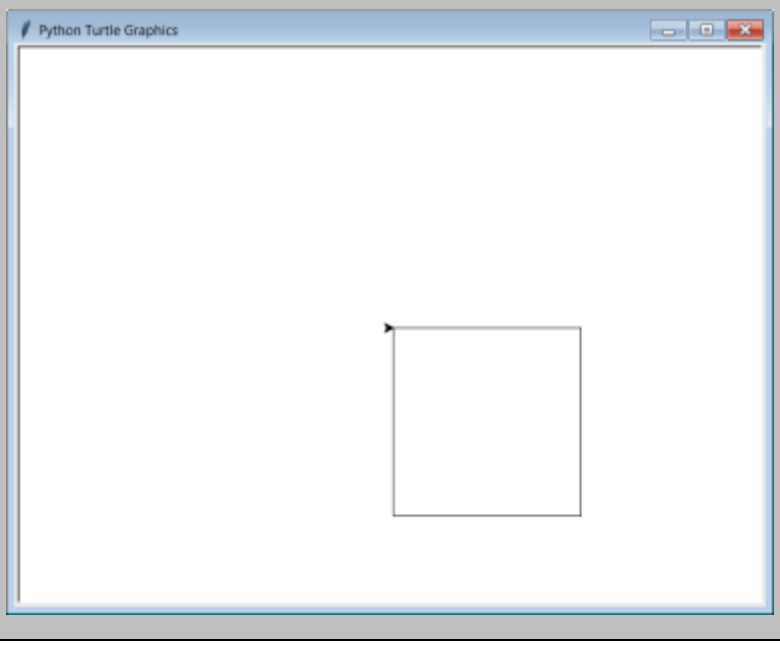
```
1 # TurtleGraphics07.py
2 # This program shows that the "turtle"
3 # can also move "backward".
4
5
6 from turtle import *
7
8 setup(800,600)
9
10 left(90)
11 forward(100)
12 right(90)
13 backward(200)
14
15 update()
16 done()
17
```



The next few programs are going to draw some shapes. Program **TurtleGraphics08.py**, in Figure 5.9, draws a square by going **forward** and turning **right** 4 times.

Figure 5.9

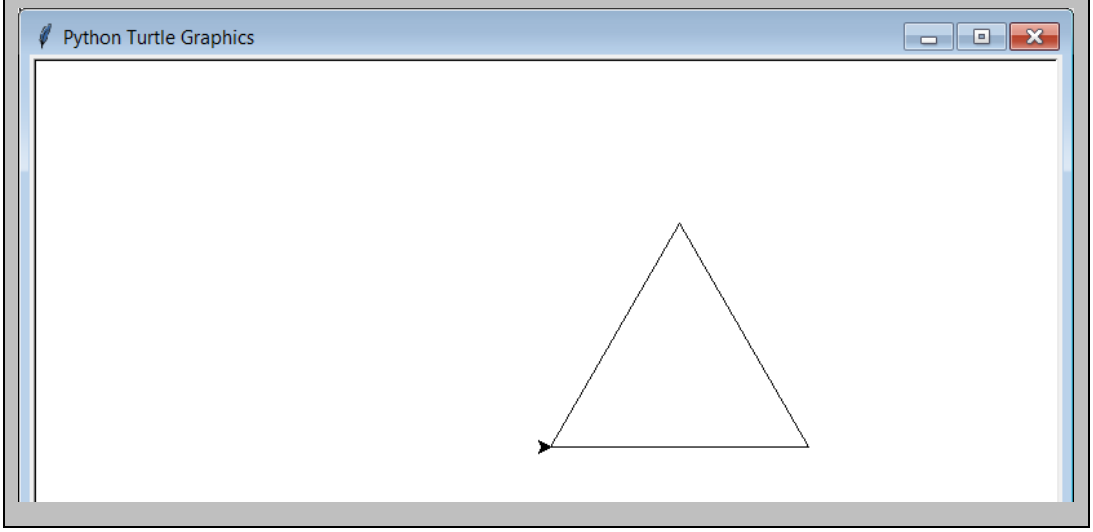
```
# TurtleGraphics08.py
# This program makes a square by going
# "forward" and turning "right" 4 times.


from turtle import *

setup(800,600)

forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)

update()
done()
```



Program **TurtleGraphics09.py**, in Figure 5.10, shows that the turtle is not limited to 90-degree turns. It can actually turn any number of degrees. In this case, 3 **left** turns of **120** degrees result in an *equilateral triangle*.

Figure 5.10

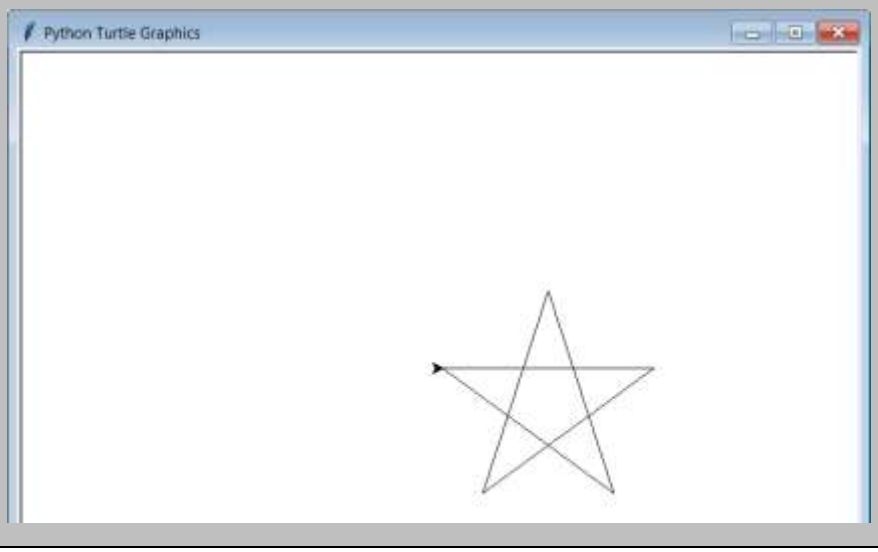
```
 1 # TurtleGraphics09.py
 2 # This program demonstrates that turns do
 3 # not need to be 90 degree "right angles".
 4 # You can turn any number of degrees.
 5 # With 3 turns of 120 degrees, I can
 6 # make an equilateral triangle.
 7
 8
 9 from turtle import *
10
11 setup(800,600)
12
13 forward(200)
14 left(120)
15 forward(200)
16 left(120)
17 forward(200)
18 left(120)
19
20 update()
21 done()
```



Program **TurtleGraphics10.py**, in Figure 5.11, draws a 5-point star by making 5 **right** turns of **144** degrees. When trying to draw stars or regular polygons, there are mathematical formulas that you can use to determine the exact number of degrees to turn; however, you may find it quicker and easier to just use simple *trial-and-error*.

Figure 5.11

```
 1 # TurtleGraphics10.py
 2 # Even stars are possible.
 3
 4
 5 from turtle import *
 6
 7 setup(800,600)
 8
 9 forward(200)
10 right(144)
11 forward(200)
12 right(144)
13 forward(200)
14 right(144)
15 forward(200)
16 right(144)
17 forward(200)
18 right(144)
19
20 update()
21 done()
```



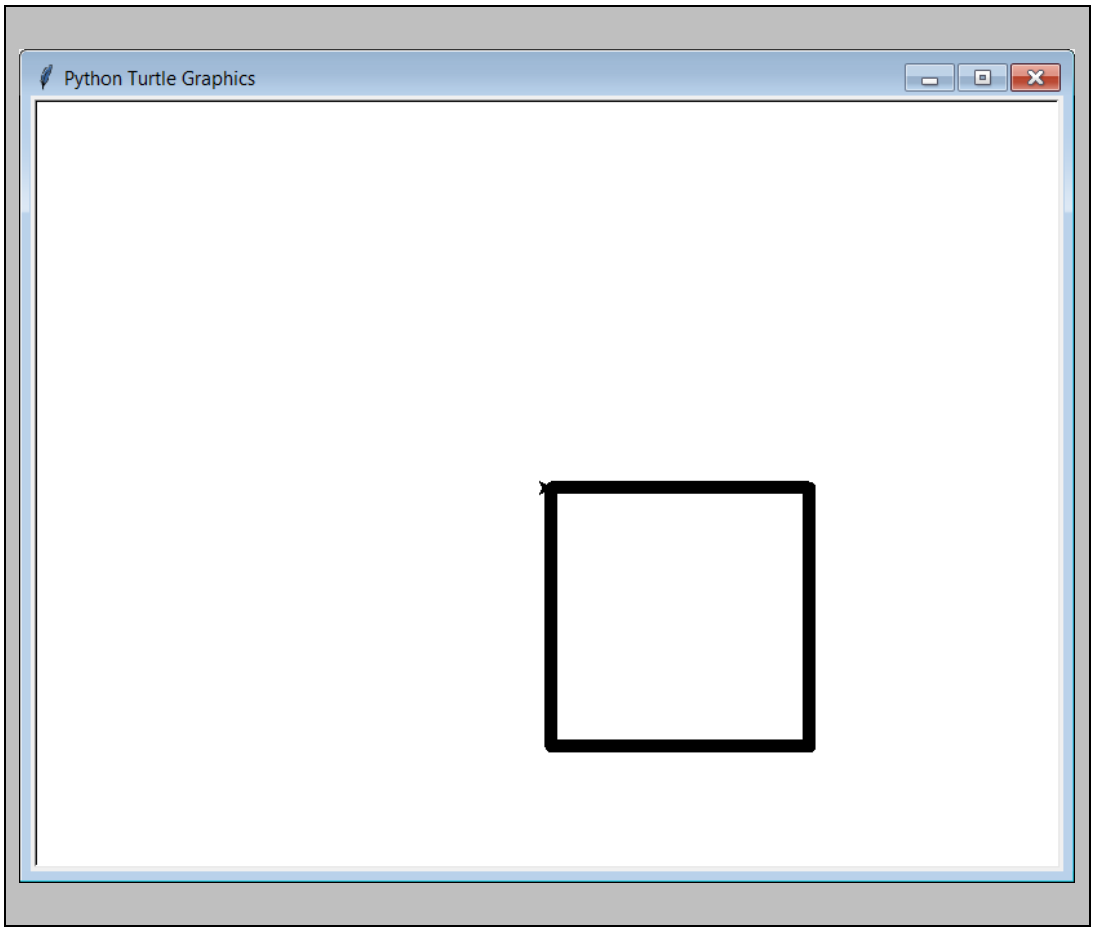
5.4 Drawing Thick or Solid Images

So far, we have only drawn either lines, or the outlines of shapes. We have not really drawn anything that is *thick* or *solid* or seems to have any substance. We can achieve this sense of "substance" in 2 different ways. One is to make the lines thicker. If we are drawing an enclosed shape, the other is to fill it in.

Program **TurtleGraphics11.py**, in Figure 5.12, demonstrates the first technique by using the command **width(10)** near the beginning. The rest of the program draws a square. The output shows that each of the 4 lines of the square is now **10** pixels thick.

Figure 5.12

```
1  # TurtleGraphics11.py
2  # This program demonstrates how you can change
3  # the thickness or <width> of the lines.
4
5
6  from turtle import *
7
8  setup(800,600)
9
10 width(10)
11 forward(200)
12 right(90)
13 forward(200)
14 right(90)
15 forward(200)
16 right(90)
17 forward(200)
18 right(90)
19
20 update()
21 done()
```

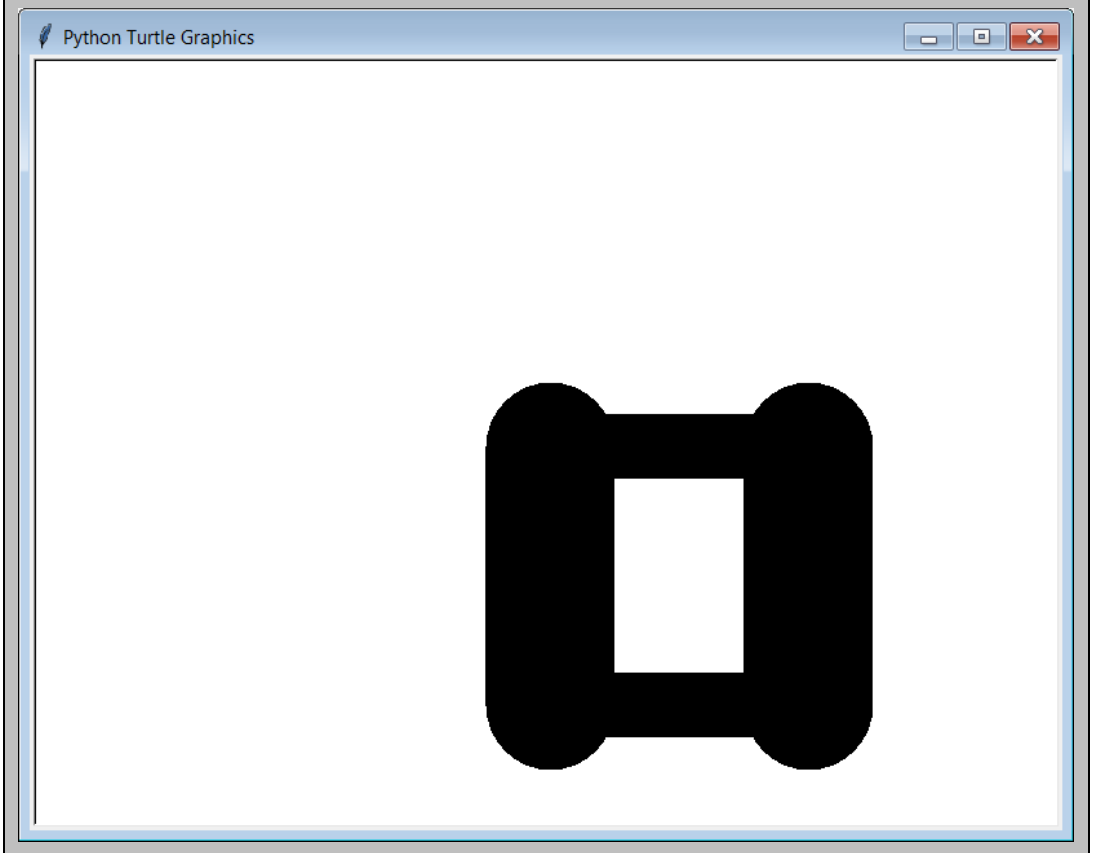


Program **TurtleGraphics12.py**, in Figure 5.13, shows that the **width** can be changed throughout the program. While this program essentially draws the same "square" as the previous program, the "width" of its 4 lines alternates between **50** and **100**.

Figure 5.13

```
1 # TurtleGraphics12.py
2 # This program demonstrates that different
3 # lines can have different widths -- which
4 # can be quite thick.
5
6
7 from turtle import *
8
9 setup(800,600)
```

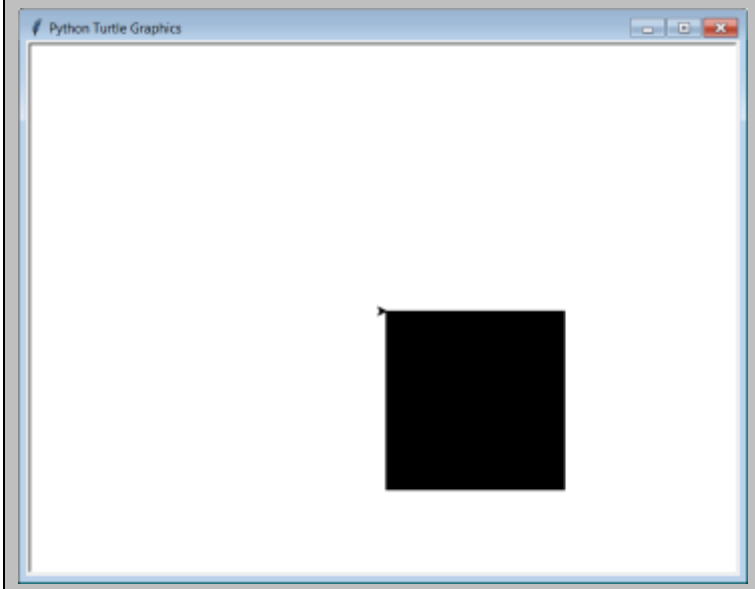
```
10
11 width(50)
12 forward(200)
13 right(90)
14 width(100)
15 forward(200)
16 right(90)
17 width(50)
18 forward(200)
19 right(90)
20 width(100)
21 forward(200)
22 right(90)
23
24 update()
25 done()
```



Program **TurtleGraphics13.py**, in Figure 5.14, demonstrates the technique of "filling in" an enclosed shape. In order to do this, we need to place the commands **begin_fill()** and **end_fill()** before and after, respectively, the code that draws the enclosed shape. In this program, lines 10 through 17 draw a square, but since this code in inside **begin_fill()** and **end_fill()** commands, we will get a solid square.

Figure 5.14

```
 1 # TurtleGraphics13.py
 2 # This program demonstrates that you can
 3 # "fill" enclosed shapes.
 4
 5 from turtle import *
 6
 7 setup(800,600)
 8
 9 begin_fill()
10 forward(200)
11 right(90)
12 forward(200)
13 right(90)
14 forward(200)
15 right(90)
16 forward(200)
17 right(90)
18 end_fill()
19
20 update()
21 done()
```



NOTE:
The square will not be "filled in" until it is completely drawn.

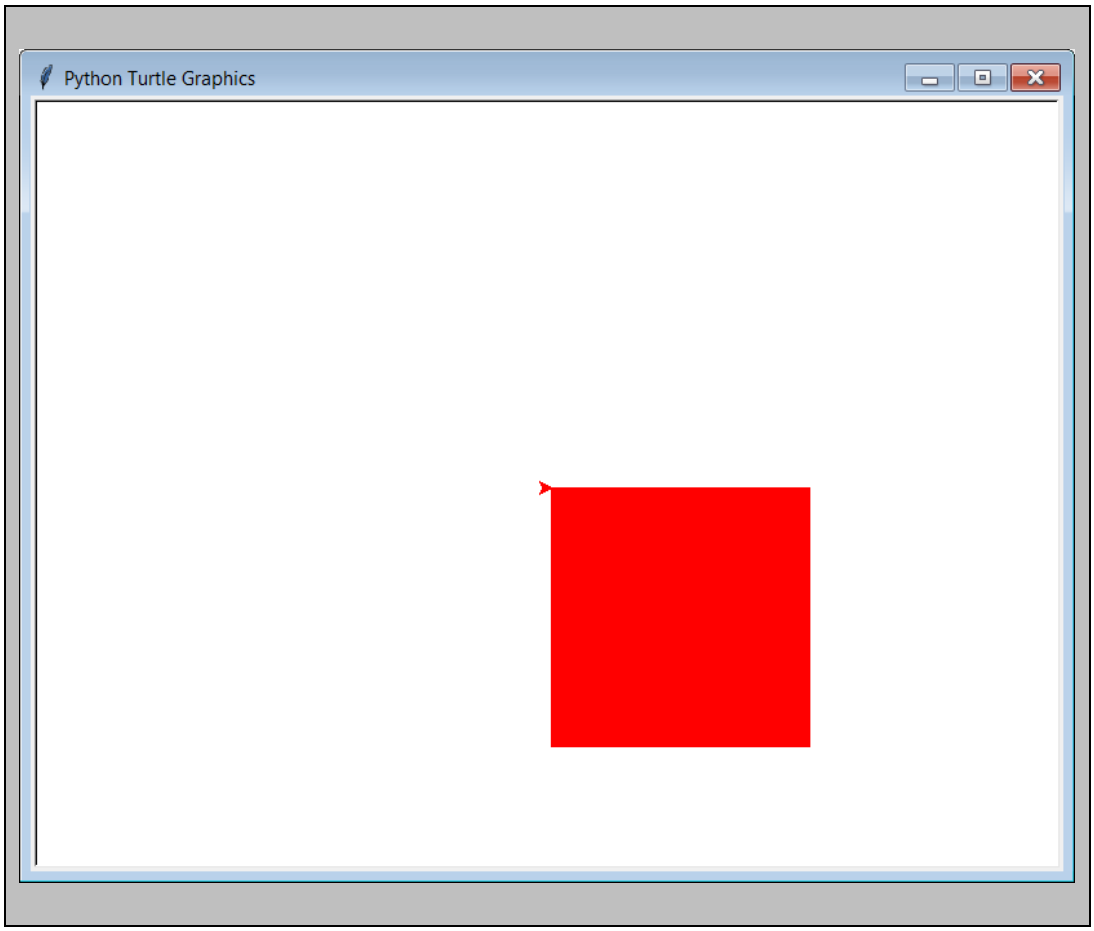
5.5 Adding Color

If all you have are the colors black and white, things can get boring rather quickly. I mean, do you know anyone who owns a Black-and-White television? Maybe a better question is, do you know anyone who *remembers* owning a Black-and-White TV? Once color TVs came out, the older black-and-white models disappeared. Why? People like color.

Program **TurtleGraphics14.py**, in Figure 5.15, is almost identical to the previous program, the one that drew a solid square. The difference is the **color("red")** command has been added near the beginning. Once you use **color**, everything following will be in that "color".

Figure 5.15

```
 1 # TurtleGraphics14.py
 2 # This program demonstrates that you can
 3 # draw in other colors besides "black".
 4
 5
 6 from turtle import *
 7
 8 setup(800,600)
 9
10 color("red")
11 begin_fill()
12 forward(200)
13 right(90)
14 forward(200)
15 right(90)
16 forward(200)
17 right(90)
18 forward(200)
19 right(90)
20 end_fill()
21
22 update()
23 done()
24
```

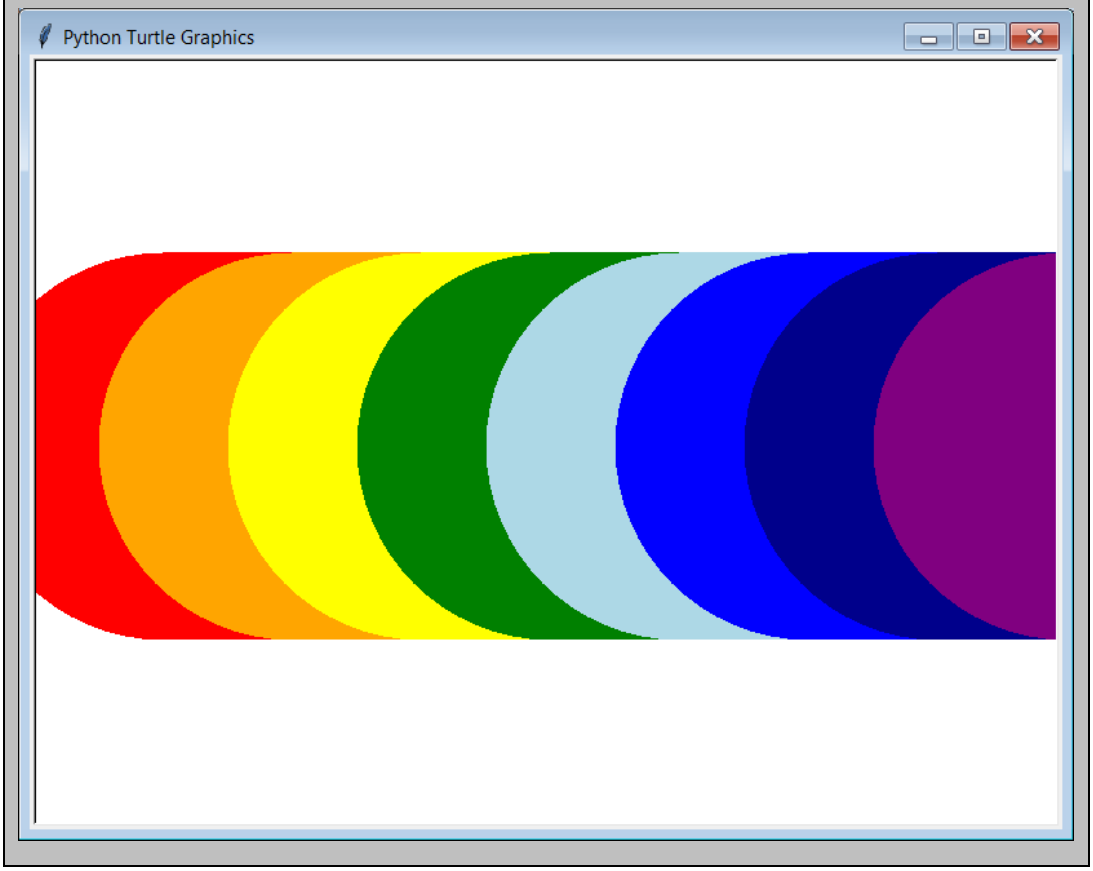


There are 140 different colors from which you can choose. Program **TurtleGraphics15.py**, in Figure 5.16, demonstrates 8 of the available colors. Note that some colors come in "dark" and "light" shades.

Figure 5.16

```
# TurtleGraphics15.py
# This program demonstrates several colors.

from turtle import *

setup(800,600)

backward(300)
width(300)
color("red")
forward(100)
```

```
12 color("orange")
13 forward(100)
14 color("yellow")
15 forward(100)
16 color("green")
17 forward(100)
18 color("light blue")
19 forward(100)
20 color("blue")
21 forward(100)
22 color("dark blue")
23 forward(100)
24 color("purple")
25 forward(100)
26
27 update()
28 done()
```



5.6 Lifting the Pen

The name of this section seems a little weird. What "pen" are we "lifting?" To truly understand its meaning, we need to look at the next program.

Program **TurtleGraphics16.py**, in Figure 5.17, is about 3 times as long as any program we have seen so far. The program attempts to draw 4 separate squares, with one in each of the 4 corners of the window. Before each square is drawn, the turtle needs to *move* to the appropriate corner.

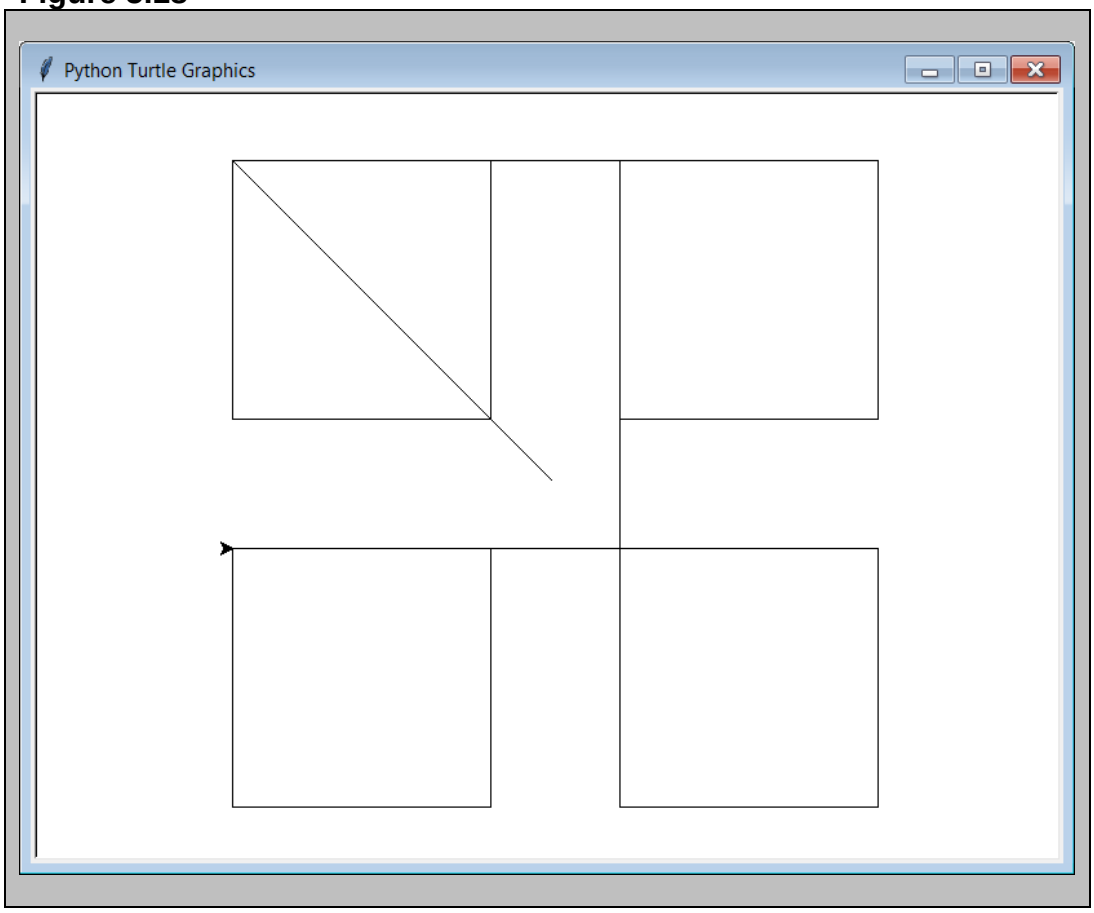
Figure 5.17

```
 1 # TurtleGraphics16.py
 2 # This program tries to draw 4 separate squares.
 3 # Before each square is drawn; the "turtle" moves
 4 # to one of the corners of the graphics window.
 5 # The problem is the turtle is always drawing
 6 # a several lines are drawn that we do not want.
 7
 8
 9 from turtle import *
10
11 setup(800,600)
12
13 left(135)     # face North-West
14 forward(350)  # Move to top-left corner
15 right(135)    # face East again
16
17 # draw square
18 forward(200)
19 right(90)
20 forward(200)
21 right(90)
22 forward(200)
23 right(90)
24 forward(200)
25 right(90)
26
27 forward(300)  # Move to top-right corner
28
29 # draw square
30 forward(200)
31 right(90)
```

```
32 forward(200)
33 right(90)
34 forward(200)
35 right(90)
36 forward(200)
37 right(90)
38
39 right(90)     # face South
40 forward(300)  # Move to bottom-right corner
41 left(90)      # face East again
42
43 # draw square
44 forward(200)
45 right(90)
46 forward(200)
47 right(90)
48 forward(200)
49 right(90)
50 forward(200)
51 right(90)
52
53 backward(300) # Move to bottom-left corner
54
55 # draw square
56 forward(200)
57 right(90)
58 forward(200)
59 right(90)
60 forward(200)
61 right(90)
62 forward(200)
63 right(90)
64
65 update()
66 done()
```

Maybe you already see the problem with the program. Maybe you don't. Look at the output in Figure 5.18 and you will definitely see the problem. While the 4 squares were drawn, it also drew the paths the turtle took to get to each of the 4 corners. This is because, by default, the "pen" attached to the turtle's tail, is "down". If you want the turtle to move somewhere, without drawing along the way, you need to first make the turtle "lift the pen". Hopefully this section's title now makes sense.

Figure 5.18



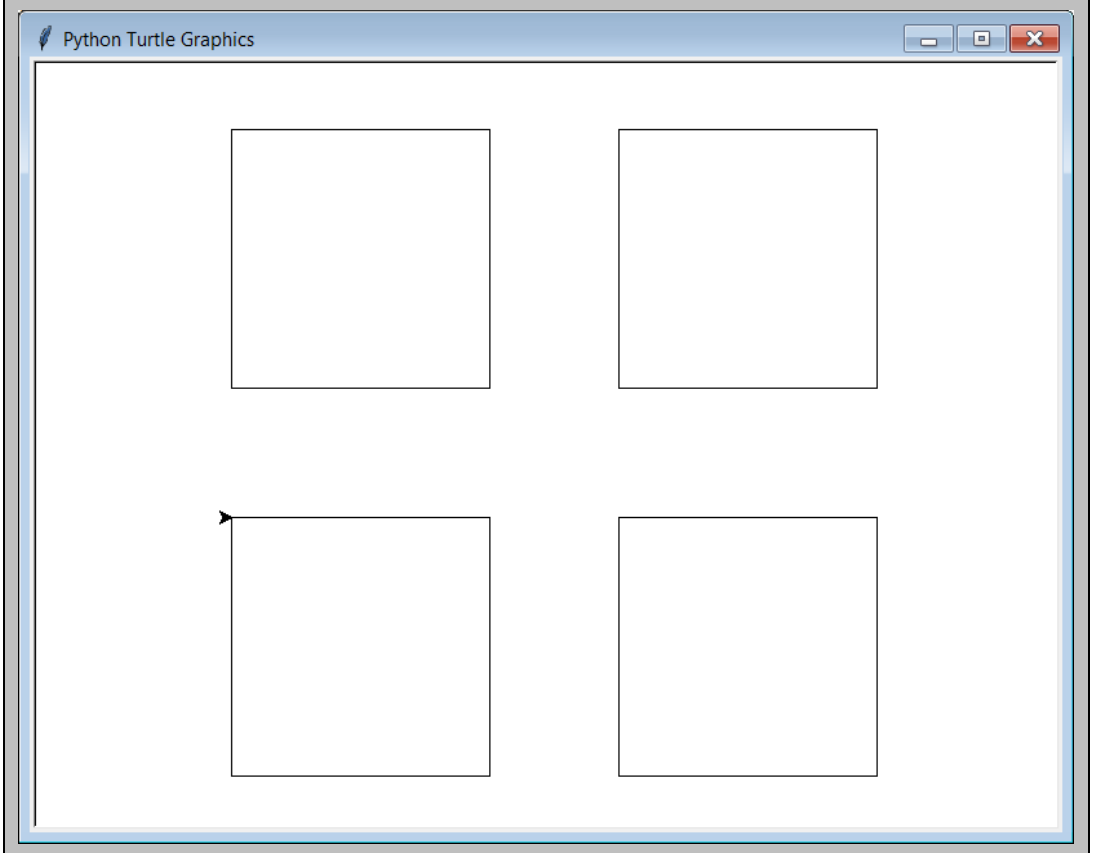
Program **TurtleGraphics17.py**, in Figure 5.19, fixes the issue of the previous program by adding strategic **penup()** and **pendown()** commands at various parts of the program. Basically, when you are moving the turtle to a specific position, the "pen" should be "up". When you actually want to draw something, the "pen" should be "down".

Figure 5.19

```
 1 # TurtleGraphics17.py
 2 # This program improves on the previous program
 3 # by adding strategic <penup> and <pendown>
 4 # commands at the appropriate places.
 5 # Now, only the 4 squares are drawn.
 6
 7
 8 from turtle import *
 9
10 setup(800,600)
11
```

```
12 penup()
13 left(135)     # face North-West
14 forward(350)  # Move to top-left corner
15 right(135)    # face East again
16 pendown()
17
18 # draw square
19 forward(200)
20 right(90)
21 forward(200)
22 right(90)
23 forward(200)
24 right(90)
25 forward(200)
26 right(90)
27
28 penup()
29 forward(300)  # Move to top-right corner
30 pendown()
31
32 # draw square
33 forward(200)
34 right(90)
35 forward(200)
36 right(90)
37 forward(200)
38 right(90)
39 forward(200)
40 right(90)
41
42 penup()
43 right(90)     # face South
44 forward(300)  # Move to bottom-right corner
45 left(90)      # face East again
46 pendown()
47
48 # draw square
49 forward(200)
50 right(90)
51 forward(200)
52 right(90)
53 forward(200)
```

```
54 right(90)
55 forward(200)
56 right(90)
57
58 penup()
59 backward(300) # Move to bottom-left corner
60 pendown()
61
62 # draw square
63 forward(200)
64 right(90)
65 forward(200)
66 right(90)
67 forward(200)
68 right(90)
69 forward(200)
70 right(90)
71
72 update()
73 done()
```



5.7 Clearing the Window

Sometimes you may want to draw something, clear the window, and then draw something else. That seems simple enough, but there are some issues involved with clearing the window that we will investigate in this final section.

Program **TurtleGraphics18.py**, in Figure 5.20, is essentially a copy of the previous program, the one that drew the 4 squares. The difference is, now a **clear()** command has been inserted after each square is drawn. Since the window is "cleared" right after each square is drawn, you only see each of the completed squares for a split second.

Figure 5.20

```
 1 # TurtleGraphics18.py
 2 # This program puts a <clear> command after each
 3 # square is drawn.  Now we only see one square
 4 # (briefly) at a time.
 5
 6 from turtle import *
 7
 8 setup(800,600)
 9
10 penup()
11 left(135)      # face North-West
12 forward(350)   # Move to top-left corner
13 right(135)     # face East again
14 pendown()
15
16 # draw square
17 forward(200)
18 right(90)
19 forward(200)
20 right(90)
21 forward(200)
22 right(90)
23 forward(200)
24 right(90)
25
26 clear()
27
28 penup()
29 forward(300)   # Move to top-right corner
30 pendown()
31
32 # draw square
```

```
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)

clear()

penup()
right(90)      # face South
forward(300)   # Move to bottom-right corner
left(90)       # face East again
pendown()

# draw square
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)

clear()

penup()
backward(300) # Move to bottom-left corner
pendown()

# draw square
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)

clear()

update()
done()
```

I cannot really display the output of this program as each square is erased the moment it is drawn. You really need to execute this program on your computer to see and appreciate the output.

What we need is some type of time delay. Something that will make the computer pause for just a second so we can see the first image drawn before the window clears and the computer draws the next image. Python has such a command. The **sleep** command will make the computer "sleep" or wait for a certain number of seconds. One thing, the **sleep** command is not part of the **turtle** library. It is part of the **time** library. No problem, we can import that library as well.

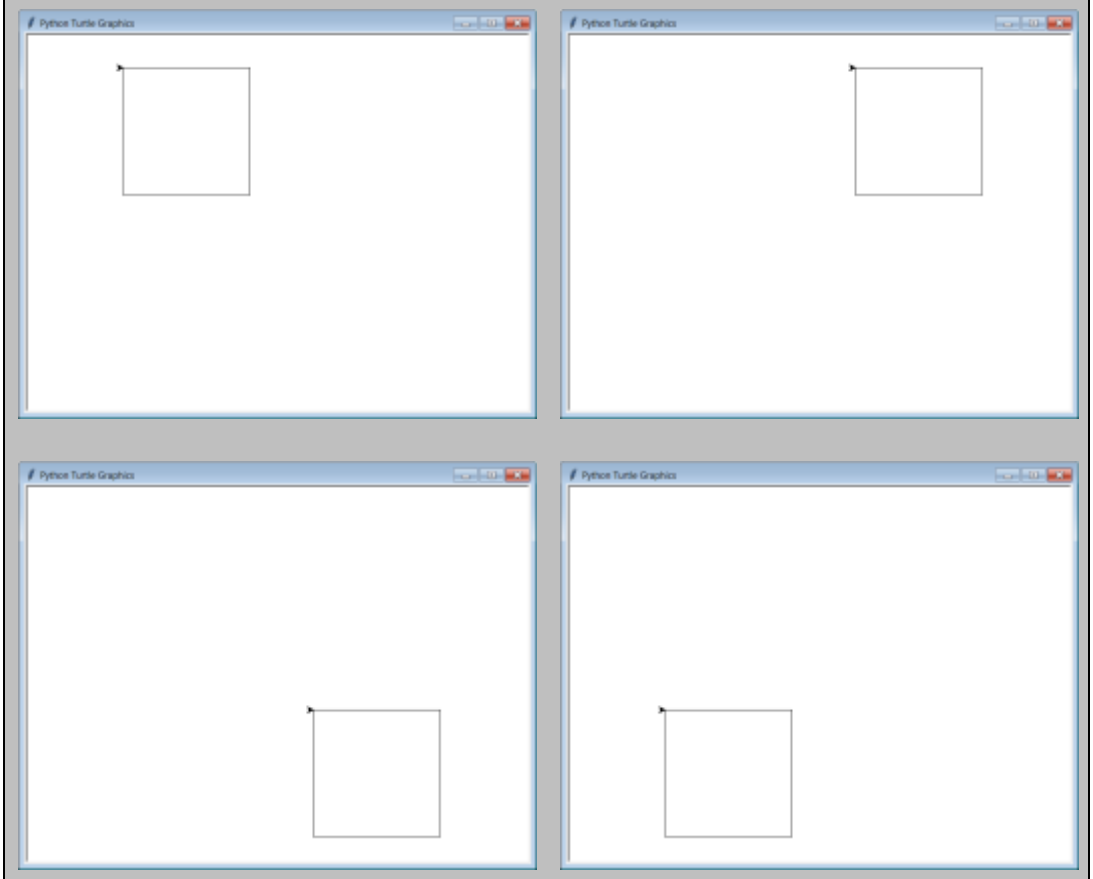
Program **TurtleGraphics19.py**, in Figure 5.21, imports the **sleep** command **from** the **time** library. Since **sleep** is the only command from the **time** library that I need, it is the only one that I will import. This is why I am not using a wildcard like I am with the **turtle** library. Now a **sleep(1)** command has been inserted before every **clear()** command. This will make the computer wait one full second before it clears the window, which will give us a chance to see each individual square.

Figure 5.20

```
# TurtleGraphics19.py
# This program imports the <sleep> command from
# the <time> library which allows you to make
# the turtle pause or "sleep" for a specified
# number of seconds.  Now each square stays on
# the screen for a full second before it is erased.


from turtle import *
from time import sleep

setup(800,600)

penup()
left(135)      # face North-West
forward(350)   # Move to top-left corner
right(135)     # face East again
pendown()

# draw square
forward(200)
right(90)
forward(200)
right(90)
```

```
forward(200)
right(90)
forward(200)
right(90)

sleep(1)
clear()

penup()
forward(300)  # Move to top-right corner
pendown()

# draw square
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)

sleep(1)
clear()

penup()
right(90)     # face South
forward(300)  # Move to bottom-right corner
left(90)      # face East again
pendown()

# draw square
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)

sleep(1)
clear()

penup()
```

```
backward(300) # Move to bottom-left corner
pendown()

# draw square
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)
forward(200)
right(90)

sleep(1)
clear()

update()
done()

```



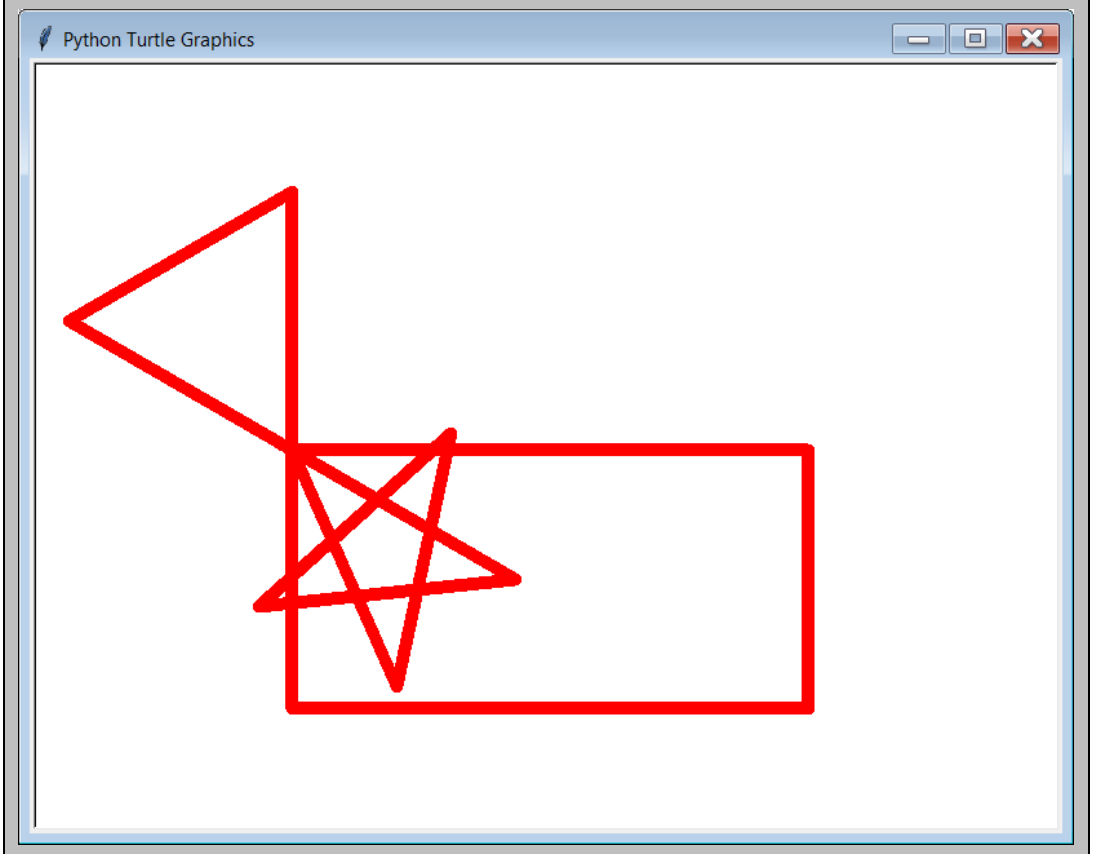
You may have the impression that the only thing of which to be concerned when clearing the window is having a delay. That is not always correct. The next 2 programs help to illustrate this.

Program **TurtleGraphics20.py**, in Figure 5.22, draws 3 shapes. Specifically, it draws a rectangle, followed by a triangle, followed by a star. We have seen triangles and stars drawn in earlier programs, but they will appear different in this program. In the earlier programs, the triangle or the star was the only thing drawn. In this program, there is something else drawn first. Look at the output and see how the conclusion of one shape affects the beginning of the next. By "beginning" I specifically mean the starting location and direction.

Figure 5.22

```
# TurtleGraphics20.py
# This program draws a rectangle, a triangle,
# and a star.   Note how the way one shape
# finishes affects the location and rotation
# of the next shape.

from turtle import *

setup(800,600)
width(10)
color("red")

# rectangle
backward(200)
forward(400)
right(90)
forward(200)
right(90)
forward(400)
right(90)
forward(200)

#triangle
forward(200)
left(120)
forward(200)
left(120)
forward(200)

# star
forward(200)
```

```
32 right(144)
33 forward(200)
34 right(144)
35 forward(200)
36 right(144)
37 forward(200)
38 right(144)
39 forward(200)
40 right(144)
41
42 update()
43 done()
```

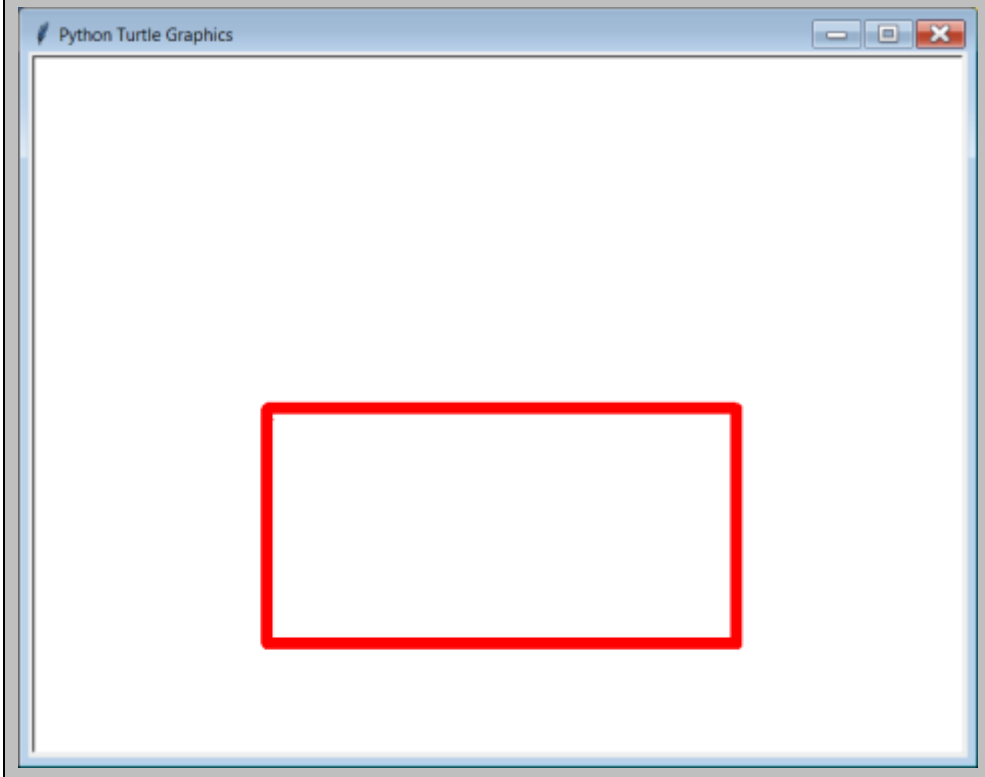


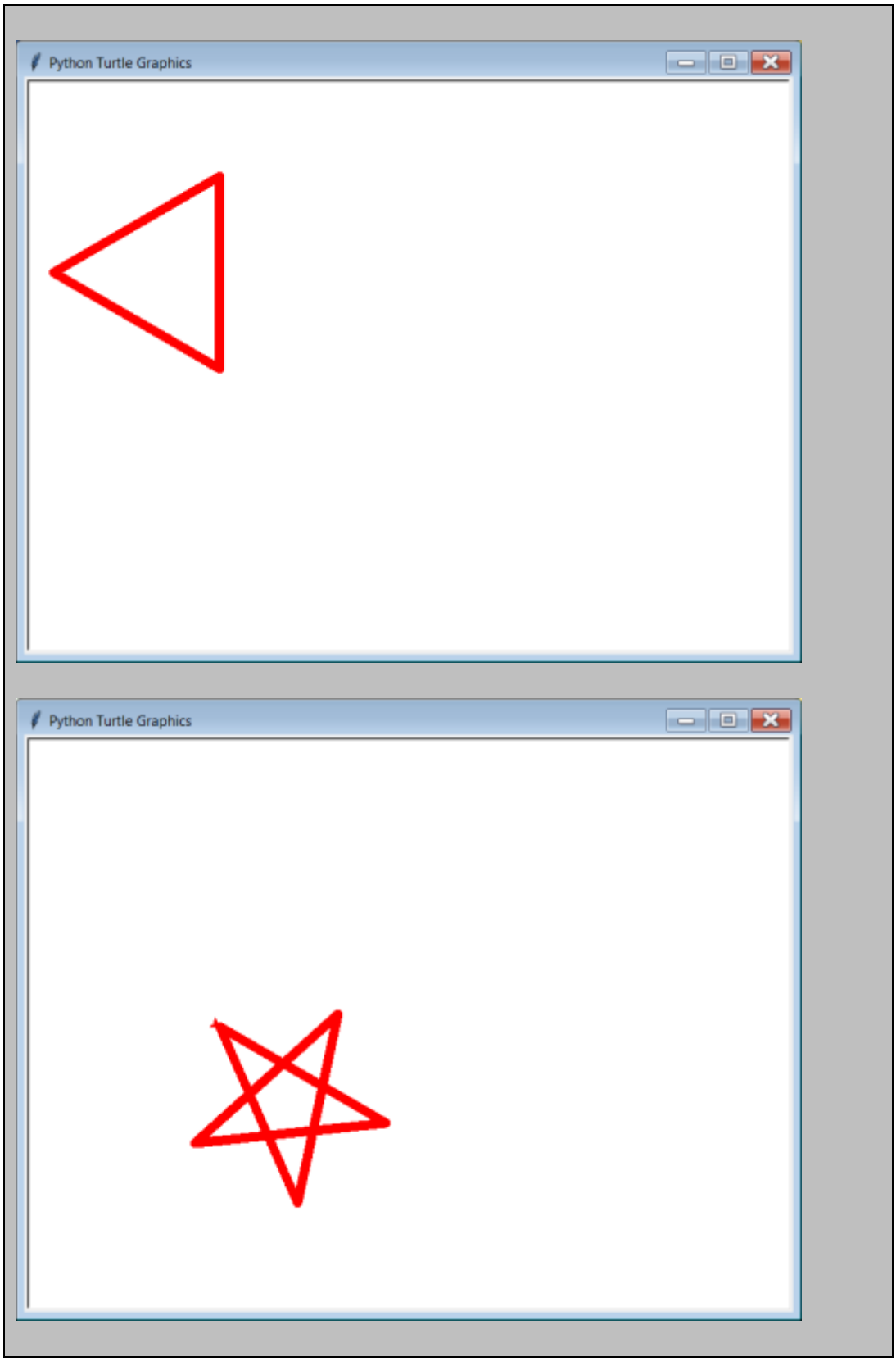
Program **TurtleGraphics21.py**, in Figure 5.23, is almost identical to the previous program. The only difference now is the window is cleared after each shape is drawn. Note that the triangle and star have the exact same location, orientation, color and width as they did in the previous program. Clearing the window made no difference in terms of where the shapes are and how they are rotated.

Figure 5.23

```
 1 # TurtleGraphics21.py
 2 # This program draws the same shapes from the
 3 # previous program, but uses a <clear> command
 4 # after each shape is drawn.  Note how clearing
 5 # the window this way has no effect on the
 6 # shape's location, orientation, color or width.
 7
 8 from turtle import *
 9 from time import sleep
10
11 setup(800,600)
12 width(10)
13 color("red")
14
15 # rectangle
16 backward(200)
17 forward(400)
18 right(90)
19 forward(200)
20 right(90)
21 forward(400)
22 right(90)
23 forward(200)
24
25 sleep(1)
26 clear()
27
28 #triangle
29 forward(200)
30 left(120)
31 forward(200)
32 left(120)
33 forward(200)
34
35 sleep(1)
36 clear()
```

```
37
38 # star
39 forward(200)
40 right(144)
41 forward(200)
42 right(144)
43 forward(200)
44 right(144)
45 forward(200)
46 right(144)
47 forward(200)
48 right(144)
49
50 sleep(1)
51 clear()
52
53 update()
54 done()
```



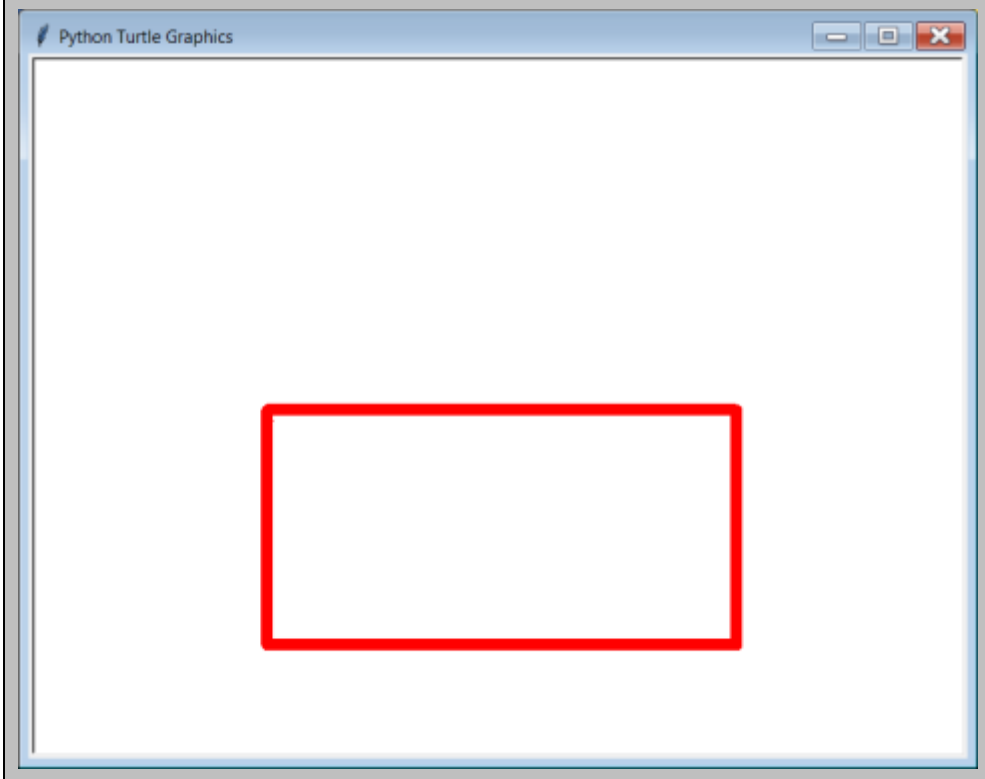

Python Turtle Graphics
Python Turtle Graphics

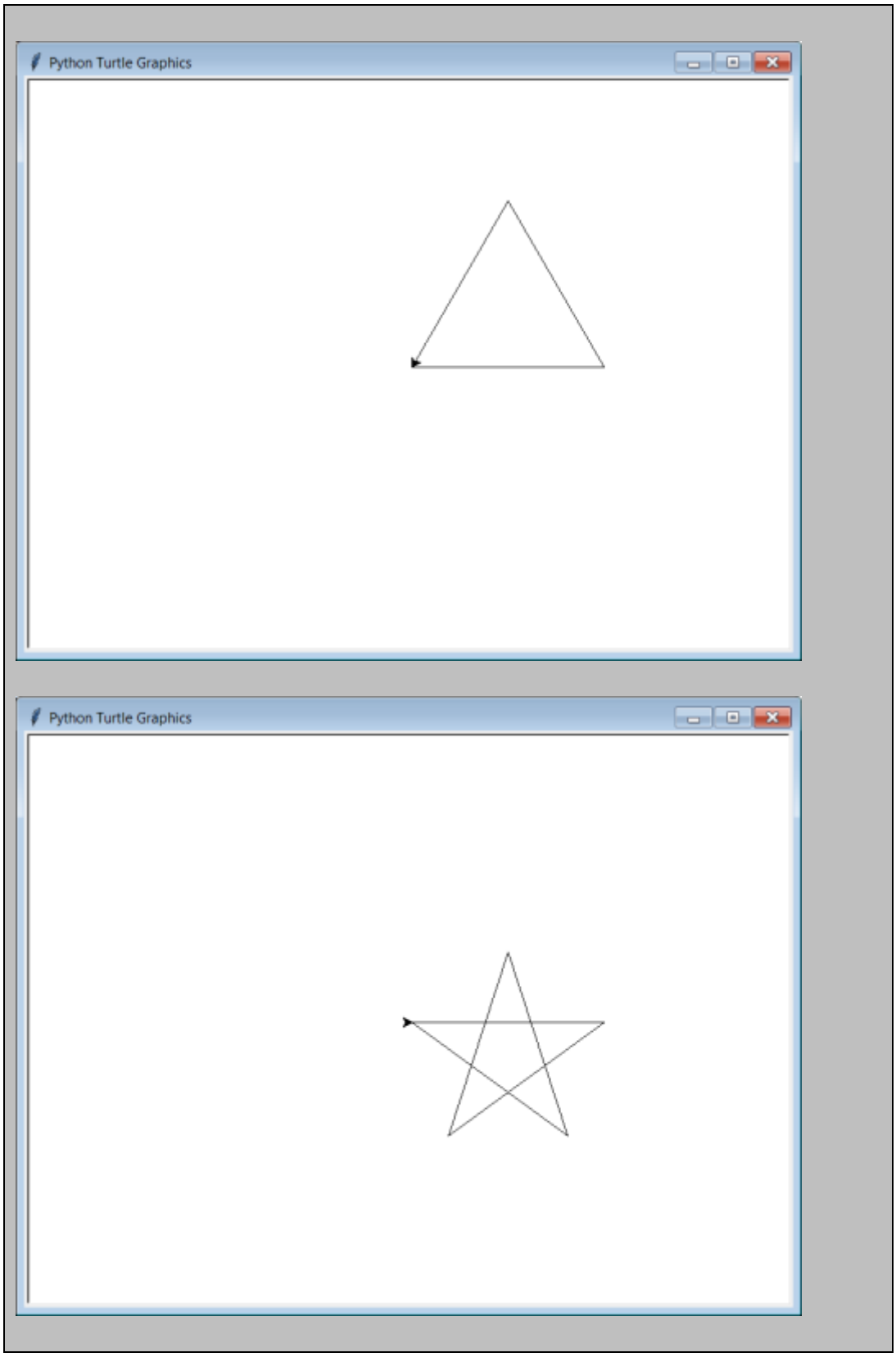
The final program of this chapter is program **TurtleGraphics22.py**, shown in Figure 5.24. This program introduces the **reset** command. **reset** is similar to **clear** in that it "clears" the window; however, that is not all that **reset** does. It basically "resets" the Turtle Graphics window to its original state. This means in addition to clearing the window, it also moves the turtle back to the center of the screen, makes it face right again and sets the **width** and **color** back to their default values of **1** and "**black**" respectively. When you look at the output, you will see that the triangle and the star are now oriented normally because all the shapes start by drawing to the right.

Figure 5.24

```
 1 # TurtleGraphics22.py
 2 # This program replaces the <clear> commands
 3 # of the previous program with <reset>.
 4 # The <reset> command does more than clear the
 5 # screen.  It also returns the turtle back to
 6 # the center of the screen and makes it face
 7 # right, sets the <width> value back to 1,
 8 # and sets the <color> value back to "black".
 9
10 from turtle import *
11 from time import sleep
12
13 setup(800,600)
14 width(10)
15 color("red")
16
17 # rectangle
18 backward(200)
19 forward(400)
20 right(90)
21 forward(200)
22 right(90)
23 forward(400)
24 right(90)
25 forward(200)
26
27 sleep(1)
28 reset()
29
30 #triangle
31 forward(200)
32 left(120)
33 forward(200)
34 left(120)
```

```
35 forward(200)
36
37 sleep(1)
38 reset()
39
40 # star
41 forward(200)
42 right(144)
43 forward(200)
44 right(144)
45 forward(200)
46 right(144)
47 forward(200)
48 right(144)
49 forward(200)
50 right(144)
51
52 sleep(1)
53 reset()
54
55 update()
56 done()
```




Python Turtle Graphics
Python Turtle Graphics

clear vs. reset	
clear	**reset**
Clears the screen only	Clears the screen and Returns the turtle to the center of the screen and Makes the turtle face right and Returns the **width** back to its default value of **1** and Returns the **color** back to its default value of **"black"**

More Turtle Graphics Commands?
There are actually many more commands available in the **turtle** graphics library. We are not going to cover those commands in this class. The reason is we want to focus more on using “Traditional Graphics” which is coordinate based. This will be introduced in the next chapter and used throughout the entire school year. This chapter on “Turtle Graphics” was simply meant to be an introduction.