Chapter XVIII

More Graphics:
Events and Animation

Chapter XVIII Topics

18.1 Introduction

We have finally come to the last chapter. This happens to be my favorite chapter, and not because it is the "last" one. In this chapter you will learn more advanced graphics features. These features will not be as advanced as those taught in AP® Computer Science-A or Advanced Graphics, but they still will allow you to create more impressive graphics programs.

Aside from being the last chapter, this chapter is special in another way. There will be no homework exercises for this chapter. There will also be no quizzes and no test. The information covered in this chapter will not even be part of the Final Exam. However, there are 2 lab assignments – 1 minor lab and 1 major lab – that are based on the information in this chapter. At John Paul II High School the major lab counts as the 4th Quarter Project.

18.2 Mouse Events

You have learned about 2 different types of subroutines. There are *functions*, which return a value, and *procedures*, which do not. There actually is a third type of subroutine. This is an *event*. Events are not "called" in the normal sense. They are *triggered* when something specific happens. When you start to incorporate *events* into your programs, you may be surprised at the sophistication of these interactive graphics programs that do not really have much programming code.

Counting Mouse Clicks

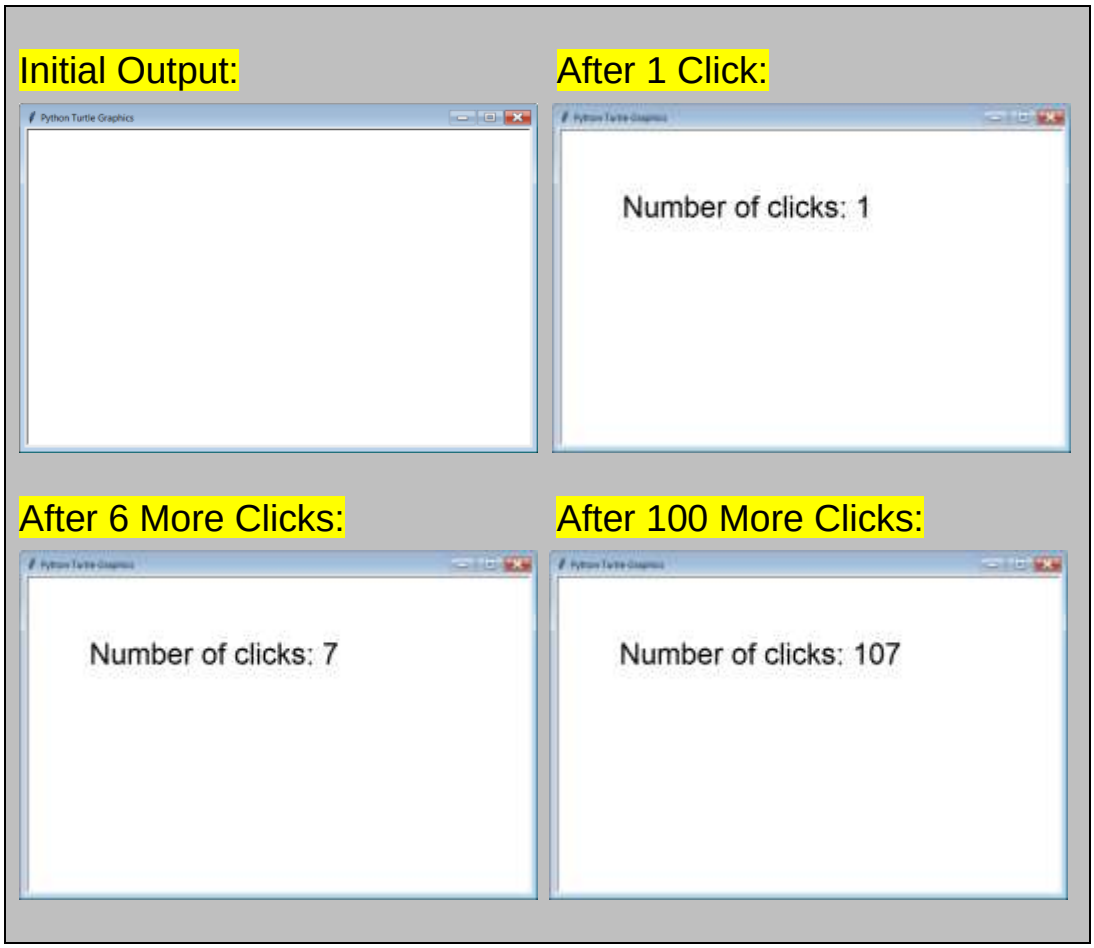
Now let us get the ball rolling with program **MouseEvents01.py**, in Figure 18.1. This program will count and display the number of times that the mouse is clicked. What makes this possible is the special **onscreenclick** event. As its name implies, this event is "triggered" anytime the user clicks on the screen, or more specifically, in the graphics window. Note that the parameter for this procedure is **display** which is the name of a procedure defined in this program.

What essentially happens is whenever the user clicks in the graphics window, **onscreenclick** is triggered, and it calls the **display** procedure.

NOTE: Any procedure called by **onsreenclick** must have 2 parameters, usually **x** and **y**, even if the procedures does not use this values. While this program example does not use the parameters, later program examples will.

Figure 18.1

```
# MouseEvents01.py
# This program introduces "Mouse Interactivity"
# with the <onscreenclick> event which is triggered
# any time the user clicks on the graphics screen
# with the left mouse button.


from Graphics import *


def display(x,y):    # Parameters (x,y) are required
   global numClicks  # even if they are not used.
   clear()
   numClicks += 1
   drawString("Number of clicks: "+str(numClicks),
100,150,"Arial",28,"normal")
   update()



##########
#  MAIN  #
##########

numClicks = 0
beginGrfx(800,500)
onscreenclick(display)
endGrfx()

```



Distiguishing Between Different Mouse Buttons

Most mice and touch pads, especially on Windows PCs, have more than one button. We can "left-click" (which is usually just called "click") or we can "right-click". On modern Macs, even though their mice and touch pads have just one button, you can still manage to "right-click" in some way. There are even mice out there with 3 buttons – which allows a "middle-click". While these mice are rare, there are several mice with a "wheel" between the 2 buttons. Not only does this wheel allow scrolling, it is "clickable" as well and counts as a "middle-click".

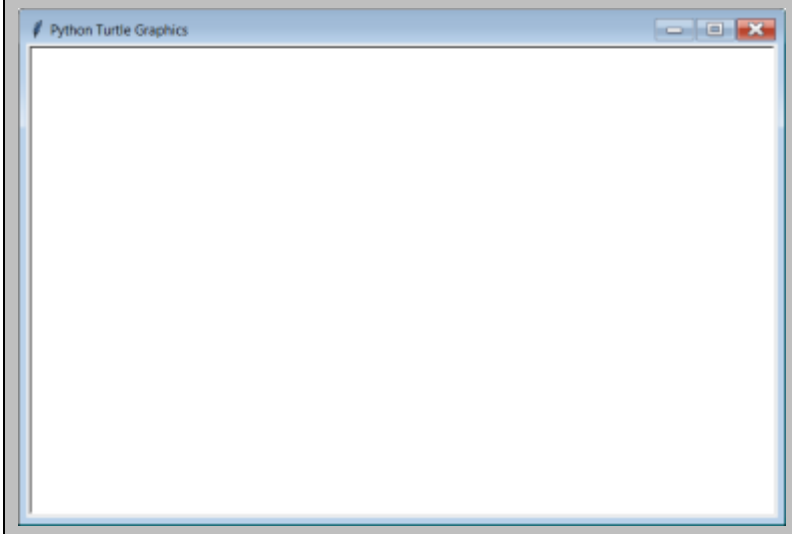
Program **MouseEvents02.py**, in Figure 18.2, is similar to the previous program, except that now instead of just counting what were essentially "left-clicks", it also counts "right-clicks" and even "middle-clicks". In order to do this, you need to add an extra argument to the **onscreenclick** event. This second argument specifies which mouse button you want. **btn=1** is the default value for this parameter and it determines the action for a "left-click". If you want to determine the action for a "right-click", you use **btn=3**. Logic would suggest that **btn=2** determines the action for a "middle-click".

Figure 18.2

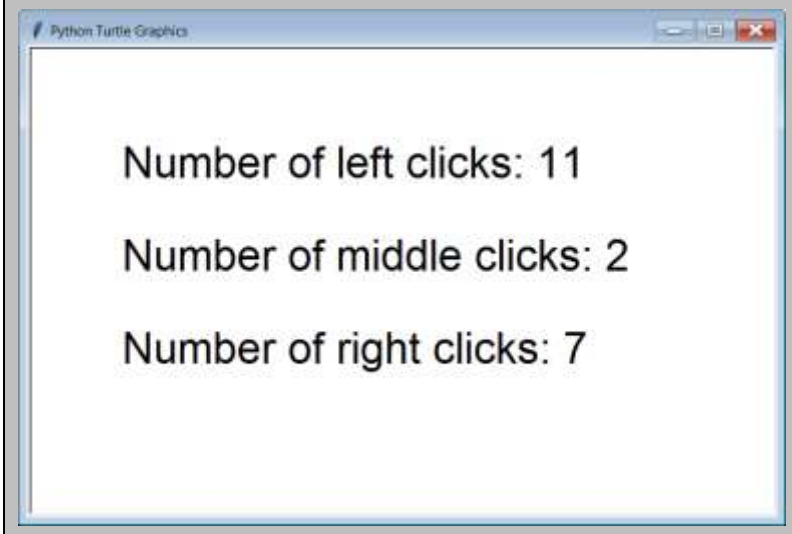
```
 1 # MouseEvents02.py
 2 # This program demonstrates how to distinguish between
 3 # the left, middle & right mouse buttons on a Windows PC.
 4 # NOTE: Some mice have a "middle" button; most do not.
 5 # ALSO: On some mice, the "wheel" is clickable and is
 6 #       actually interpreted as the middle button.
 7
 8
 9 from Graphics import *
10
11
12 def leftDisplay(x,y):
13     global numLeftClicks
14     clear()
15     numLeftClicks += 1
16     drawString("Number of left clicks: "
+str(numLeftClicks),100,150,"Arial",28,"normal")
17     drawString("Number of middle clicks: "
+str(numMiddleClicks),100,250,"Arial",28,"normal")
18     drawString("Number of right clicks: "
+str(numRightClicks),100,350,"Arial",28,"normal")
19
20
21 def middleDisplay(x,y):
22     global numMiddleClicks
23     clear()
24     numMiddleClicks += 1
25     drawString("Number of left clicks: "
+str(numLeftClicks),100,150,"Arial",28,"normal")
26     drawString("Number of middle clicks: "
+str(numMiddleClicks),100,250,"Arial",28,"normal")
27     drawString("Number of right clicks: "
+str(numRightClicks),100,350,"Arial",28,"normal")
28
29
30 def rightDisplay(x,y):
31     global numRightClicks
32     clear()
33     numRightClicks += 1
34     drawString("Number of left clicks: "
+str(numLeftClicks),100,150,"Arial",28,"normal")
35     drawString("Number of middle clicks: "
+str(numMiddleClicks),100,250,"Arial",28,"normal")
36     drawString("Number of right clicks: "
+str(numRightClicks),100,350,"Arial",28,"normal")
37
38
39
```

```
40 ##########
41 #  MAIN  #
42 ##########
43
44 numLeftClicks = 0
45 numMiddleClicks = 0
46 numRightClicks = 0
47 beginGrfx(800,500)
48 onscreenclick(leftDisplay,btn=1)
49 onscreenclick(middleDisplay,btn=2)
50 onscreenclick(rightDisplay,btn=3)
51 endGrfx()
```

Initial Output:



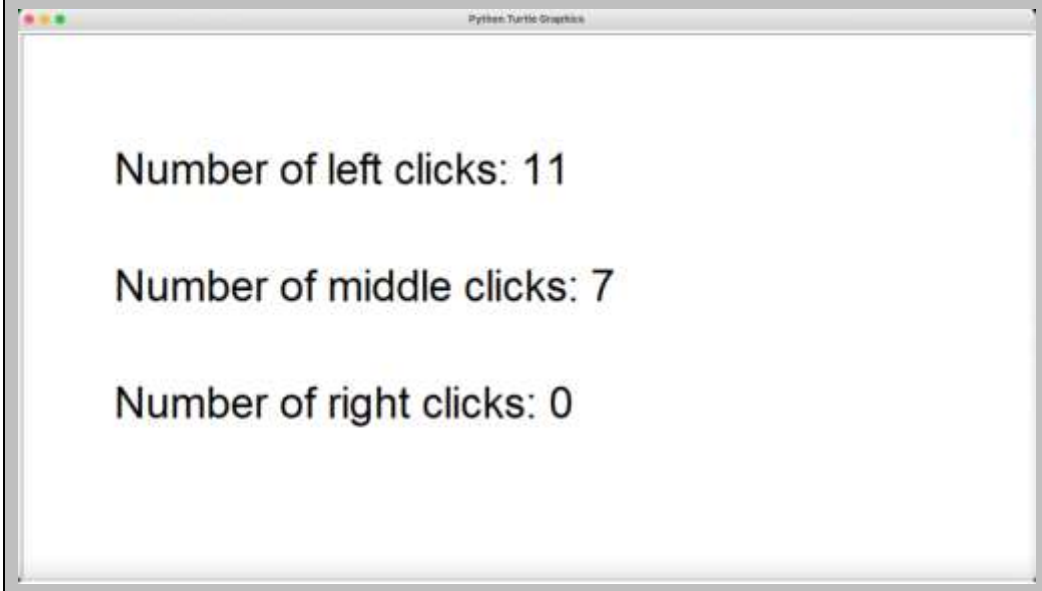
PC output after 11 left-clicks, 7 right-clicks and 2 middle-clicks:



OK, everything seemed fine, but the executions shown were from a Windows PC. When the same program is executed on a Mac, things are different. For one thing, Macs do not have a "middle button". However, Figure 18.3 shows a bigger issue. Right-clicking or "2-finger clicking" on a Mac registers as a *middle-click*.

Figure 18.3

Mac output after 11 left/1-finger-clicks and 7 right/2-finger-clicks:



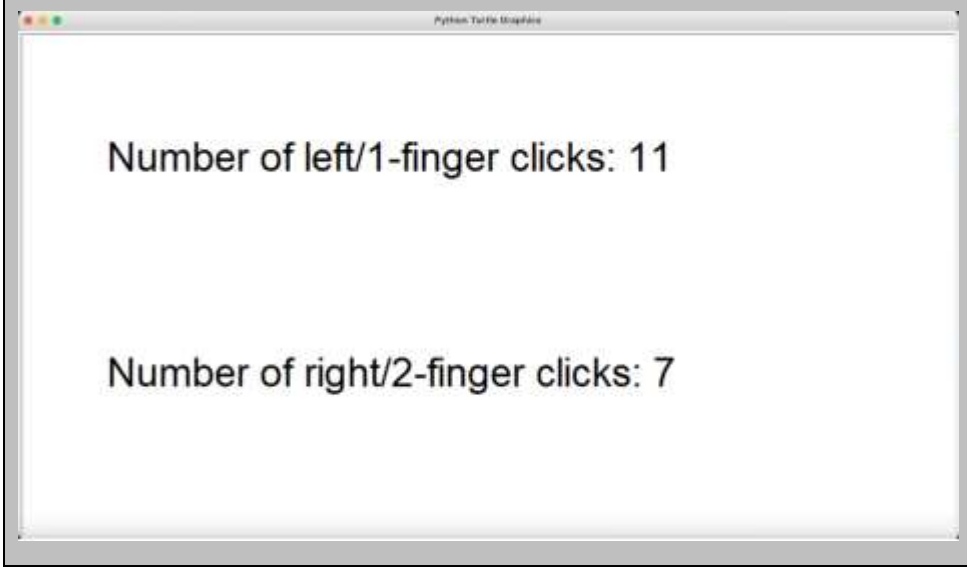
The problem is, on a Mac, the right/2-finger click has a button value of **2** instead of **3**. That is why this program does not work properly on a Mac. While more than 99% (not an exaggeration) of the programs in this textbook work on both a Mac and a PC, there are some programs that only work properly on a specific platform. Program **MouseEvents03.py**, in Figure 18.4, is basically a copy of the previous program with 2 changes. First, all the "middle-click" stuff is deleted since Macs do not have a middle-click button. Second, the button value for the right/2-finger click has been changed from **3** to **2**.

Figure 18.4

```
 1 # MouseEvents03.py
 2 # This program demonstrates how to distinguish between a
 3 # left/1-finger click & a right/2-finger click on a Mac.
 4 # NOTE: On a Mac, button 2 is for right/2-finger click.
 5 # ALSO: There is no middle-click on a Mac.
 6
 7
 8 from Graphics import *
 9
10
```

```
11 def leftDisplay(x,y):
12     global numLeftClicks
13     clear()
14     numLeftClicks += 1
15     drawString("Number of left/1-finger clicks: "
+str(numLeftClicks),100,150,"Arial",28,"normal")
16     drawString("Number of right/2-finger clicks: "
+str(numRightClicks),100,350,"Arial",28,"normal")
17
18
19 def rightDisplay(x,y):
20     global numRightClicks
21     clear()
22     numRightClicks += 1
23     drawString("Number of left/1-finger clicks: "
+str(numLeftClicks),100,150,"Arial",28,"normal")
24     drawString("Number of right/2-finger clicks: "
+str(numRightClicks),100,350,"Arial",28,"normal")
25
26
27
28 ##########
29 #  MAIN  #
30 ##########
31
32 numLeftClicks = 0
33 numRightClicks = 0
34 beginGrfx(800,500)
35 onscreenclick(leftDisplay,btn=1)
36 onscreenclick(rightDisplay,btn=2)
37 endGrfx()
```

Mac output after 11 left/1-finger-clicks and 7 right/2-finger-clicks:



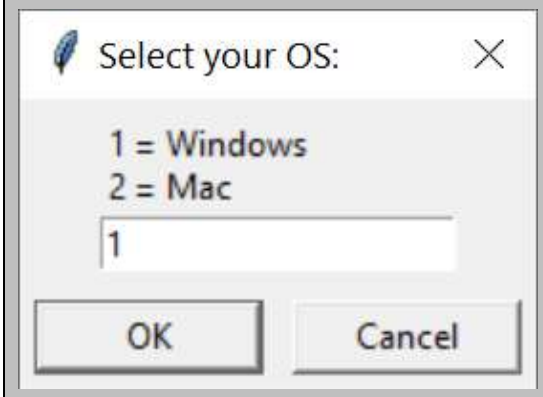
Determining the Operating System

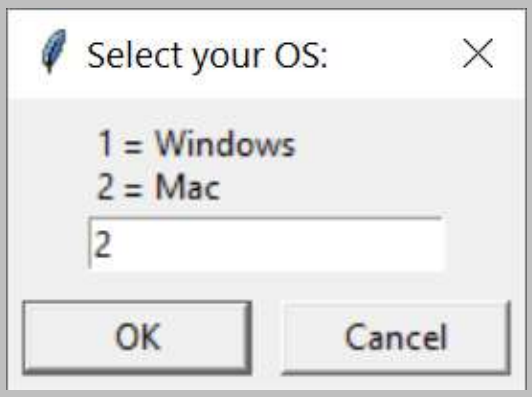
Now, it may be great to have one program for Windows PCs and another for Macs. It would be even better if one program could work for both. Program **MouseEvents04.py**, in Figure 18.5, adds a couple new variables: **right** and **middle**. You will note that these variables are used in two of the **onscreenclick** commands instead of the fixed numbers **2** and **3**. When the program is executed, the user is asked to select their operating system. If the user enters **1** for "Windows", variables **middle** and **right** are assigned new values of **2** and **3** respectively. If the user enters **2** for "Mac", the **right** variable is assigned a new values of **2** and **numMiddleClicks** is reassigned a value of **"N/A"**.

Figure 18.5

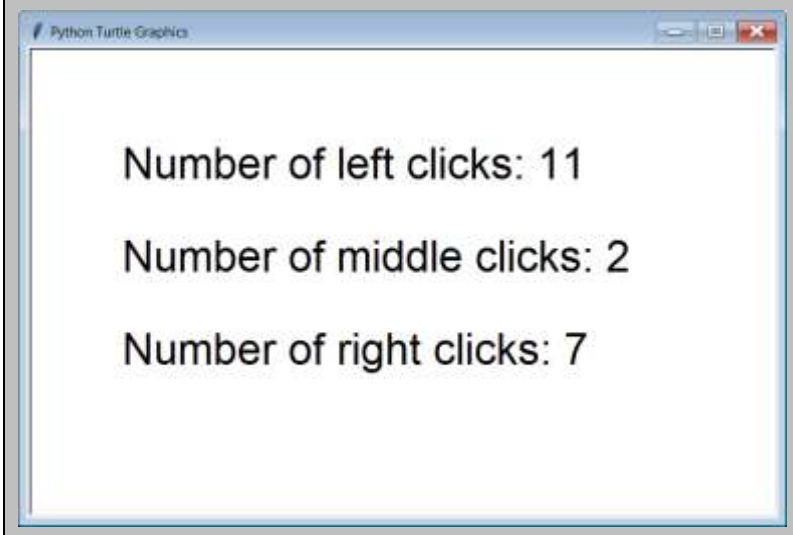
```
# MouseEvents04.py
# This program will allow the user to select the computer's
# operating system and use this information to properly
# determine the button value of the right/2-finger click
# and whether or not a middle-click exists.
# NOTE: There is no guarantee that the user will select
#       the correct operating system.


from Graphics import *


def leftDisplay(x,y):
    global numLeftClicks
    clear()
    numLeftClicks += 1
    drawString("Number of left clicks:
"+str(numLeftClicks),100,150,"Arial",28,"normal")
    drawString("Number of middle clicks:
"+str(numMiddleClicks),100,250,"Arial",28,"normal")
    drawString("Number of right clicks:
"+str(numRightClicks),100,350,"Arial",28,"normal")

def middleDisplay(x,y):
    global numMiddleClicks
    clear()
    numMiddleClicks += 1
    drawString("Number of left clicks:
"+str(numLeftClicks),100,150,"Arial",28,"normal")
    drawString("Number of middle clicks:
"+str(numMiddleClicks),100,250,"Arial",28,"normal")
```

```
27    drawString("Number of right clicks:
"+str(numRightClicks),100,350,"Arial",28,"normal")
28
29 def rightDisplay(x,y):
30    global numRightClicks
31    clear()
32    numRightClicks += 1
33    drawString("Number of left clicks:
"+str(numLeftClicks),100,150,"Arial",28,"normal")
34    drawString("Number of middle clicks:
"+str(numMiddleClicks),100,250,"Arial",28,"normal")
35    drawString("Number of right clicks:
"+str(numRightClicks),100,350,"Arial",28,"normal")
36
37
38
39 ##########
40 #  MAIN  #
41 ##########
42
43 numLeftClicks = 0
44 numMiddleClicks = 0
45 numRightClicks = 0
46 right = 4
47 middle = 4
48
49 os = numinput("Select your OS:",
                  "1 = Windows\n2 = Mac")
50 if os == 1 : # Windows
51    middle = 2
52    right = 3
53 if os == 2 : # Mac
54    right = 2
55    numMiddleClicks = "N/A"
56
57 beginGrfx(800,500)
58 onscreenclick(leftDisplay,btn=1)
59 onscreenclick(middleDisplay,btn=middle)
60 onscreenclick(rightDisplay,btn=right)
61 endGrfx()
```

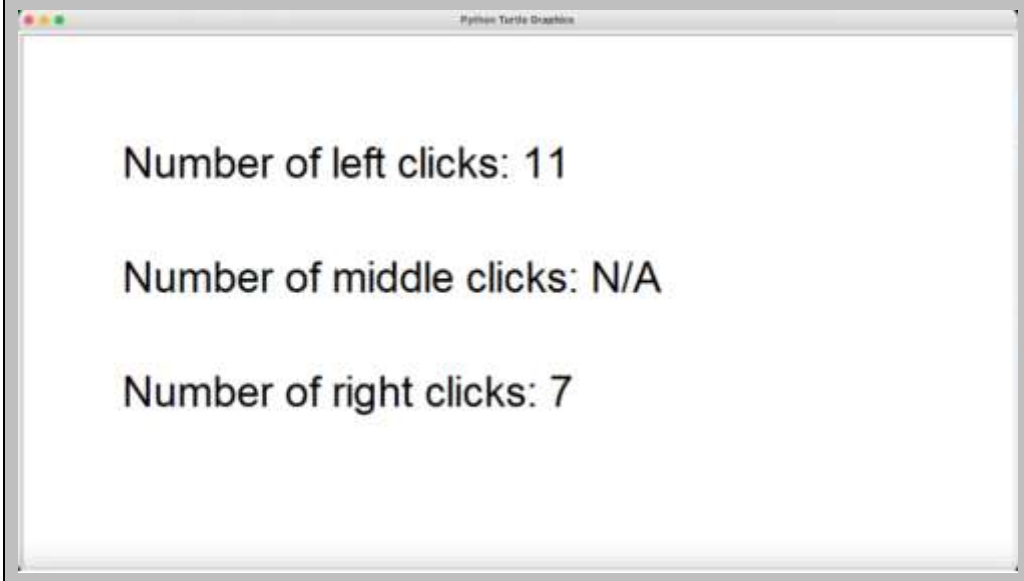




PC output after 11 left-clicks, 7 right-clicks and 2 middle-clicks:



Mac output after 11 left/1-finger-clicks and 7 right/2-finger-clicks:



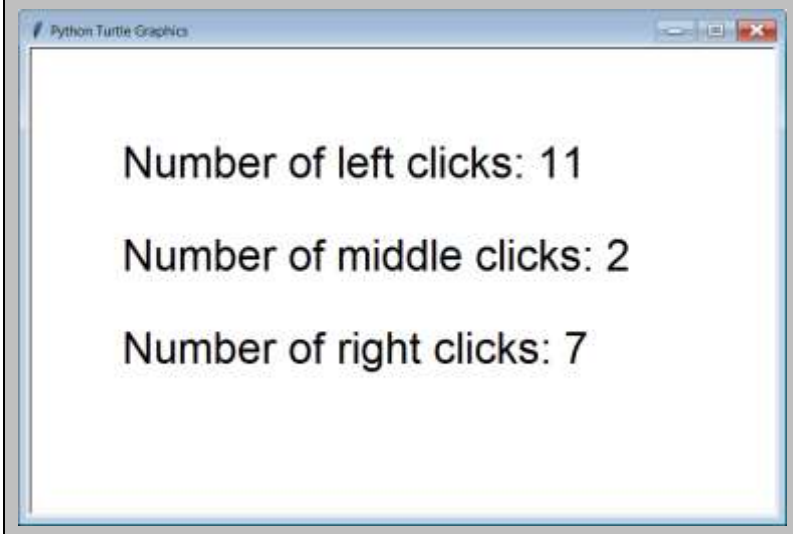
One potential issue with this program is it relies on the user to properly select the operating system. What would be even better is if the computer could detect its own operating system. This is precisely what is done by program **MouseEvents05.py**, shown in Figure 18.6. When the **os** library is imported, you can use **os.name** to determine the opeating system based on the value that it stores. A value of **"nt"** indicates a Windows Operating System. A value of **"posix"** indicates a Macintosh Operating System. Based on this, the proper values are stored in the same variables as before and the program now works without requiring the user to select their operating system.

Figure 18.6

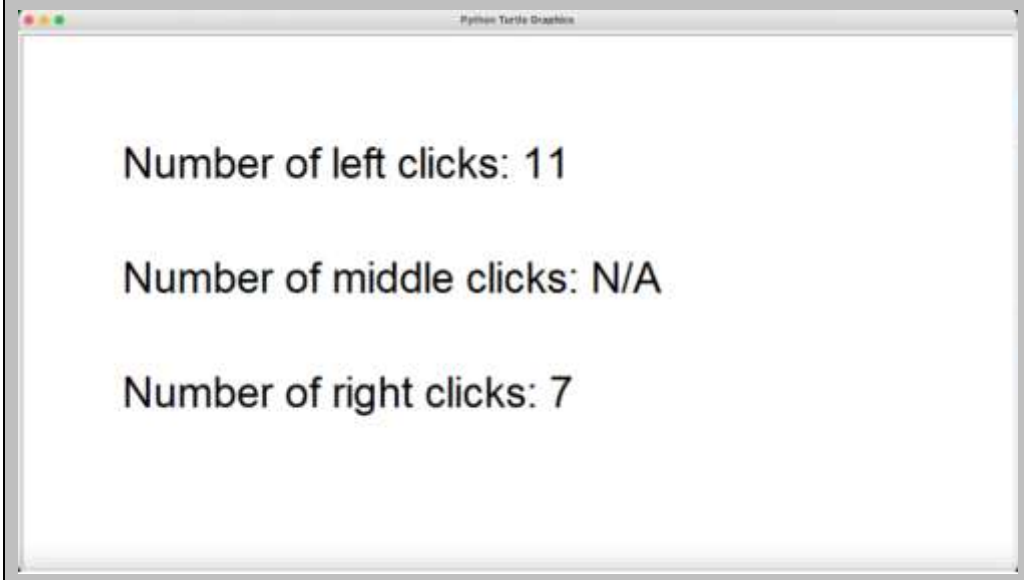
```
# MouseEvents05.py
# This program is more efficient and reliable that the
# previous program because it can actually detect the
# computer's operating system and use this information
# to properly determine the button value of the right/
# 2-finger click and whether or not a middle-click exists.


from Graphics import *
import os


def leftDisplay(x,y):
    global numLeftClicks
    clear()
    numLeftClicks += 1
    drawString("Number of left clicks:
"+str(numLeftClicks),100,150,"Arial",28,"normal")
    drawString("Number of middle clicks:
"+str(numMiddleClicks),100,250,"Arial",28,"normal")
    drawString("Number of right clicks:
"+str(numRightClicks),100,350,"Arial",28,"normal")

def middleDisplay(x,y):
    global numMiddleClicks
    clear()
    numMiddleClicks += 1
    drawString("Number of left clicks:
"+str(numLeftClicks),100,150,"Arial",28,"normal")
    drawString("Number of middle clicks:
"+str(numMiddleClicks),100,250,"Arial",28,"normal")
    drawString("Number of right clicks:
"+str(numRightClicks),100,350,"Arial",28,"normal")
```

```
28
29 def rightDisplay(x,y):
30     global numRightClicks
31     clear()
32     numRightClicks += 1
33     drawString("Number of left clicks:
"+str(numLeftClicks),100,150,"Arial",28,"normal")
34     drawString("Number of middle clicks:
"+str(numMiddleClicks),100,250,"Arial",28,"normal")
35     drawString("Number of right clicks:
"+str(numRightClicks),100,350,"Arial",28,"normal")
36
37
38
39 ##########
40 #  MAIN  #
41 ##########
42
43 numLeftClicks = 0
44 numMiddleClicks = 0
45 numRightClicks = 0
46 right = 4
47 middle = 4
48 if os.name == "nt": # Windows
49     middle = 2
50     right = 3
51 if os.name == "posix": # Mac
52     right = 2
53     numMiddleClicks = "N/A"
54
55 beginGrfx(800,500)
56 onscreenclick(leftDisplay,btn=1)
57 onscreenclick(middleDisplay,btn=middle)
58 onscreenclick(rightDisplay,btn=right)
59 endGrfx()
```

PC output after 11 left-clicks, 7 right-clicks and 2 middle-clicks:



Mac output after 11 left/1-finger-clicks and 7 right/2-finger-clicks:



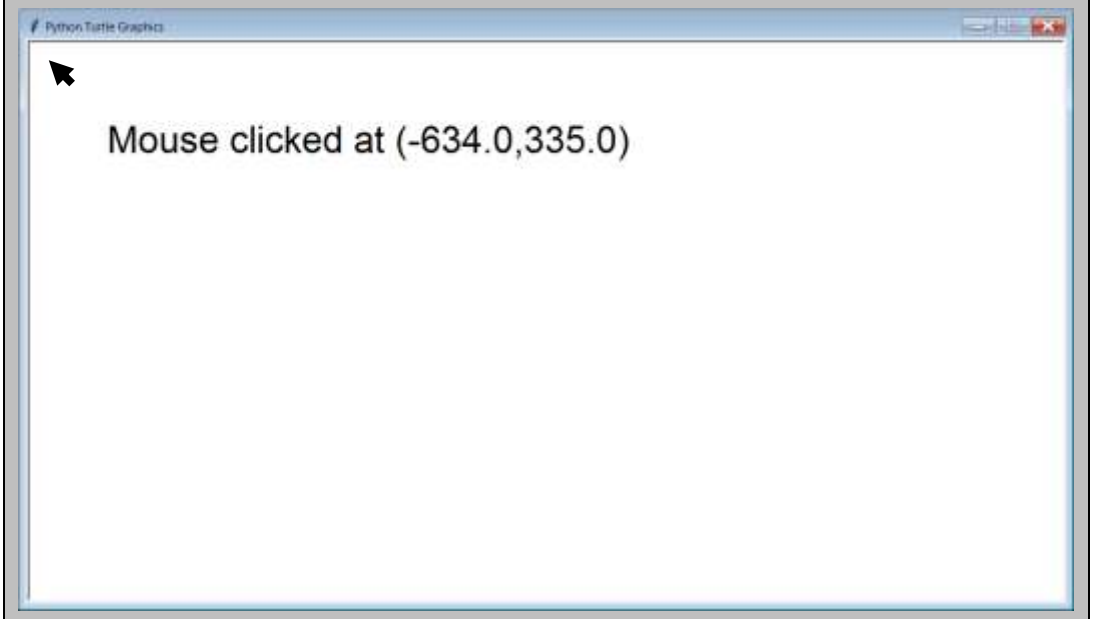
Detecting Mouse Click Location

Remember that it was mentioned that any procedure that is called from the **onscreenclick** event requires 2 parameters, usually **x** and **y**, even if those parameters are not used? What are those parameters for? We certainly did not need them in the previous 2 programs. Program **MouseEvents06.py**, in Figure 18.7, will finally make use of these 2 parameters. When you click in the graphics window, not only does **onscreenclick** call the method that you specified, it also passes the **x** and **y** values of the exact coordinate where you clicked in the window. There is one potential problem with the program. The coordinate returned is based on the coordinate system used by *Turtle Graphics* where (0,0) is in the center of the window. Any **y** values above and **x** values to the left will be negative as the outputs show.

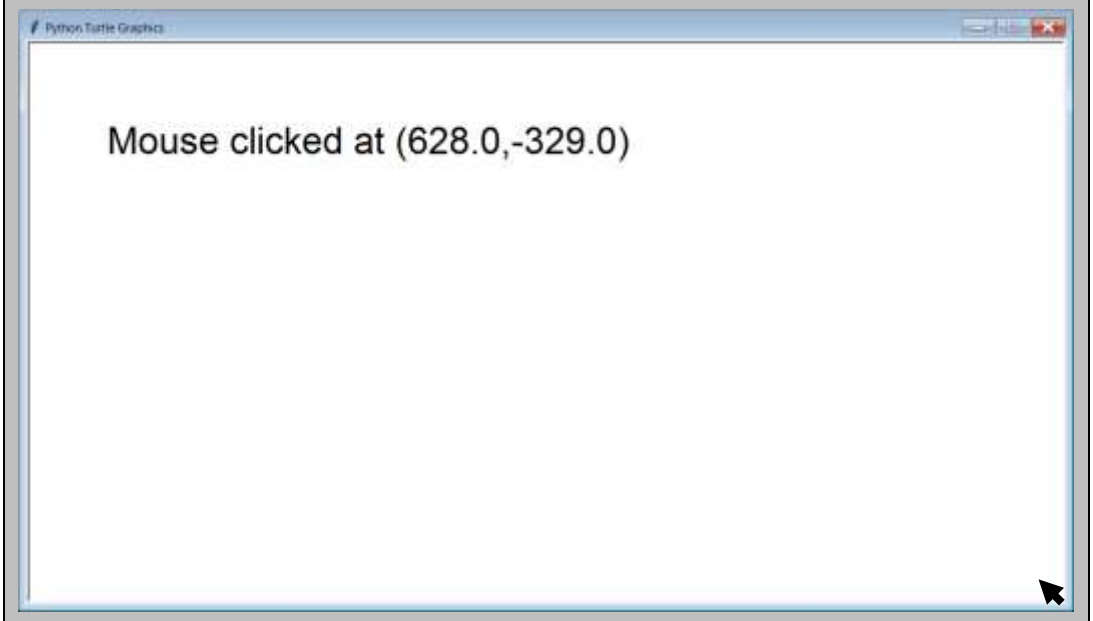
Figure 18.7

```
 1 # MouseEvents06.py
 2 # This program demonstrates that the <x> and <y>
 3 # parameters store the location where the user
 4 # clicked on the screen.  The problem is, the
 5 # coordinate displayed is based on "Turtle Graphics"
 6 # where (0,0) is in the center of the screen.
 7 # This results in some negative coordinates.
 8
 9
10 from Graphics import *
11
12
13 def display(x,y):
14     clear()
15     drawString("Mouse clicked at ("+str(x)+","+str(y)
+")",100,150,"Arial",28,"normal")
16
17
18
19 ##########
20 #  MAIN  #
21 ##########
22
23 beginGrfx(1300,700)
24 onscreenclick(display)
25 endGrfx()
26
```

Output after clicking near the top-left corner of the window:



Output after clicking near the bottom-right corner of the window:



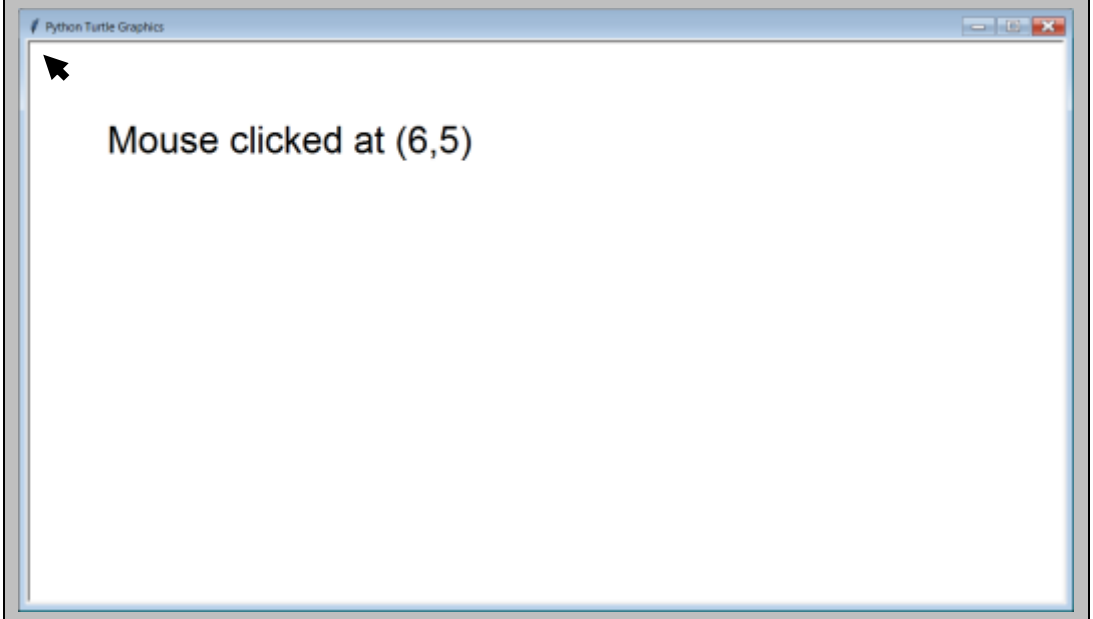
Program **MouseEvents07.py**, in Figure 18.8, fixes the issue of the previous program by using 2 *functions* from the **Graphics** library. You may have thought that the **Graphics** library only had procedures, but actually it has a few functions as well. Function **traditionalXY** will convert **x** and **y** values respectively from *Turtle Graphics*, where (0,0) is in the center of the graphics window, to *Traditional Graphics*, where (0,0) is at the top-left corner of the graphics window.

Now the output is more what we expect. When we click near the top-left corner of the graphics window, we get something close to (0,0). When we clear near the bottom-right corner of a graphics window with dimensions **1300** by **700**, we get something close to (1300,700). All coordinate values are positive now.

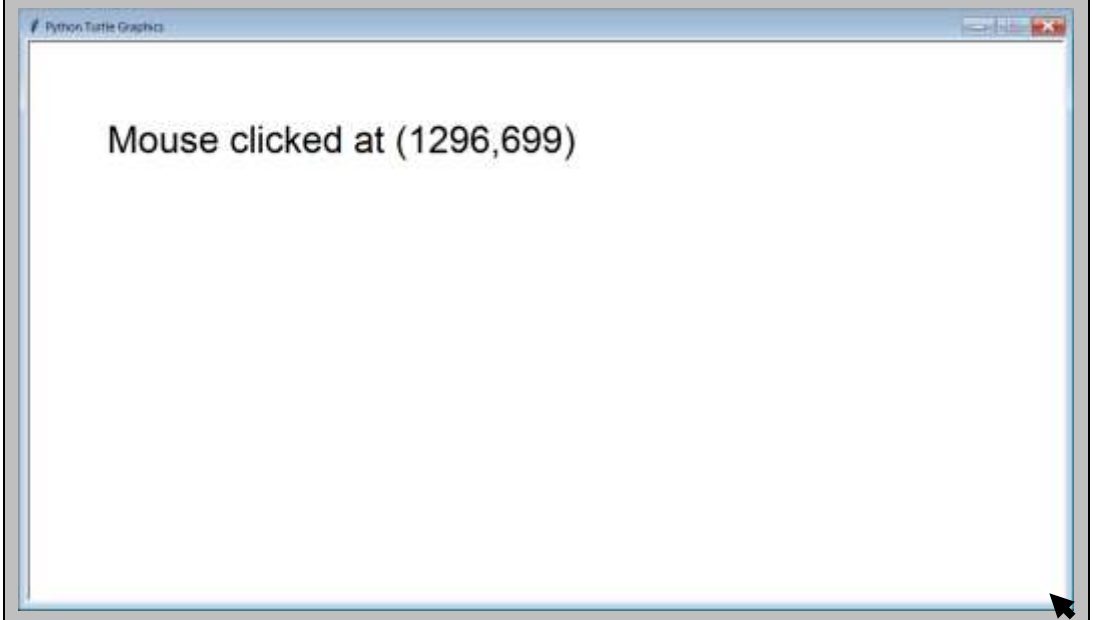
Figure 18.8

```
 1 # MouseEvents07.py
 2 # This program fixes the issue of the previous program
 3 # by using the <traditionalXY> function of the <Graphics>
 4 # library.  The result is the coordinate is converted to
 5 # "Traditional Graphics" where (0,0) is in the top-left
 6 # corner and all coordinate values are positive.
 7
 8
 9 from Graphics import *
10
11
12 def display(x,y):
13     x,y = traditionalXY(x,y)
14     clear()
15     drawString("Mouse clicked at ("+str(x)+","+str(y)+")",
100,150,"Arial",28,"normal")
16
17
18
19 ##########
20 #  MAIN  #
21 ##########
22
23 beginGrfx(1300,700)
24 onscreenclick(display)
25 endGrfx()
26
```

Output after clicking near the top-left corner of the window:



Output after clicking near the bottom-right corner of the window:



18.3 Creating Simple Paint Program Tools

Paint Program Tools vs. Entire Paint Program

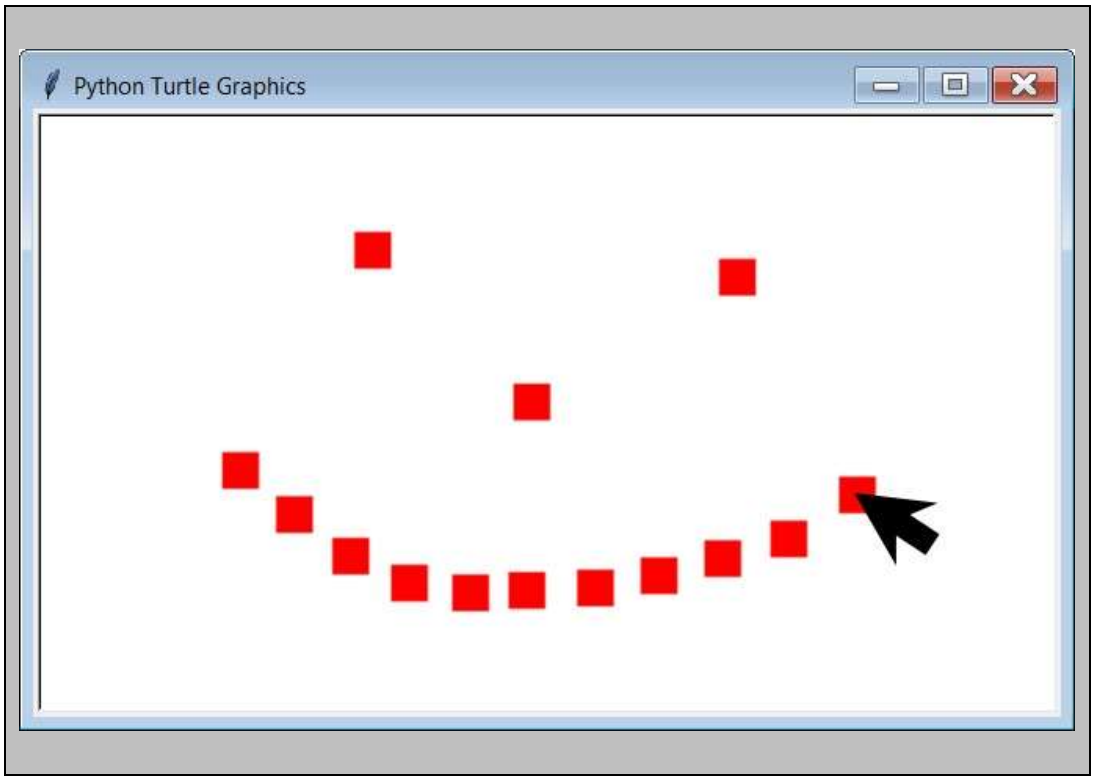
We are NOT creating an entire paint program by any stretch of the imagination. That assignment will have to wait for the *Advanced Graphics* class. In this class, we are merely introducing some concepts by creating a couple of the simpler tools. This will require the use of **onscreenclick**.

For our first program consider this: The simplest paint program would simply give you the ability to draw *dots* on the screen. Program **MouseEvents08.java**, in Figure 18.9, does just that. Remember how the previous program displayed the exact **x** and **y** values of the coordinate where the mouse was clicked. This program goes one step further. Instead of merely displaying that coordinate, it draws a "dot" or small square centered at that location.

NOTE: While it would have made sense to use **drawPoint**, I wanted the "dot" to be bigger so it would be more visible. This is why I used **fillRectangle** instead.

Figure 18.9

```
# MouseEvents08.py
# This program draws a small red
# square at every click location.


from Graphics import *


def display(x,y):
    x,y = traditionalXY(x,y)
    fillRectangle(x-7,y-7,x+7,y+7)
    update()



##########
#  MAIN  #
##########

beginGrfx(1300,700)
setColor("red")
onscreenclick(display)
endGrfx()

```



Drawing Lines

Another feature that paint programs typically have is the ability to draw straight lines. This may sound simple, but making a proper *line drawing tool* in a paint program is actually fairly complicated. Again, this is something that will wait for the *Advanced Graphics* class. For now we are going to do something much simpler. You may suspect that we would use the **drawLine** command to make this work. That seems perfectly logical and it very much can work, but the program still would be more complicated than I would like it to be. There is actually a command for drawing lines that is even simpler than **drawLine**. This is the **goto** command from the **turtle** library.

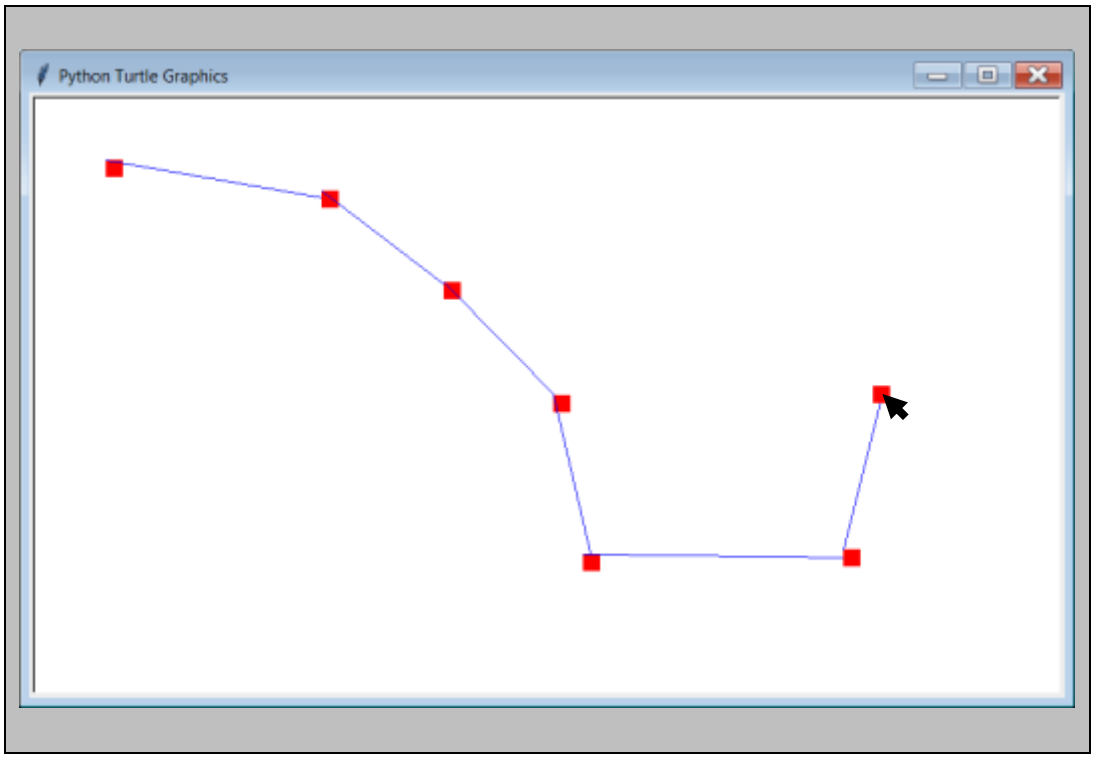
NOTE: Do not confuse the **goto** command from the **turtle** library, which "goes to" a specific coordinate on the graphics window, with the **GOTO** command used in older, unstructured languages like BASIC, which "goes to" a specific line number and resulted in *spaghetti programming*.

Unlike the **drawLine** command which requires 4 integer parameters, **goto** only requires 2. You may wonder how this is possible. A line is the connection of 2 coordinate points, and each of these points has an **x** value and a **y** value. This is true. With **goto**, you only specify the *end point*. The *start point* is simply the last coordinate that was used. If the program has just started, the *start point* is (0,0).

Something else that makes **goto** simpler for this program is that it uses the coordinates from *Turtle Graphics*, which means we will not need to convert them. Program **MouseEvents09.java**, in Figure 18.10, builds upon the previous program. While that "paint program" simply drew dots, this one will also draw lines. Left-clicking still gives you the dots. Right-clicking will draw lines using the **goto** command. Note how I used this to draw a picture of the Big Dipper.

Figure 18.10

```
# MouseEvents09.py
# This program draws a small red square when
# the mouse is left-clicked and a blue line
# when the mouse in right-clicked.


from Graphics import *


def displayDot(x,y):
    setColor("red")
    x,y = traditionalXY(x,y)
    fillRectangle(x-7,y-7,x+7,y+7)
    update()

def displayLine(x,y):
    setColor("blue")
    goto(x,y)
    update()



##########
#  MAIN  #
##########

beginGrfx(1300,700)
onscreenclick(displayDot,btn=1)
onscreenclick(displayLine,btn=3)
endGrfx()

```



Lab 18A will actually involve making a simplistic paint program. Before you can begin that lab, you still need to learn more about both Mouse and Key Events.

18.4 Creating Clickable Areas

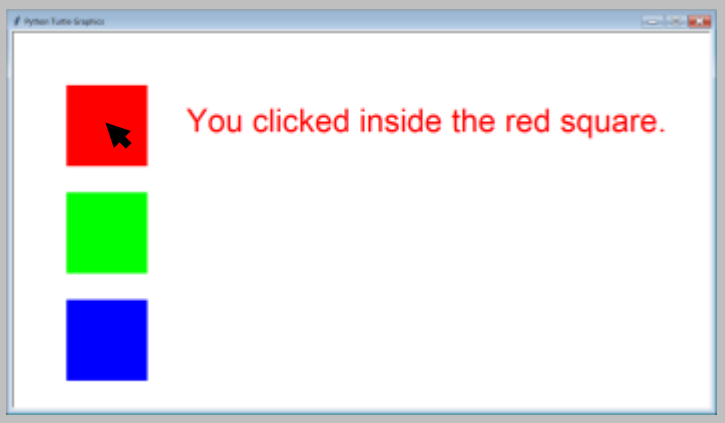
There is something that computer mouse users take for granted. It is the ability of the program to know that a mouse has clicked inside some area. To select a color in a paint program we click on some icon. To maximize or minimize a window, we click the "Window Management Buttons" in the title bar. In both cases some action results from clicking somewhere in a specified area.

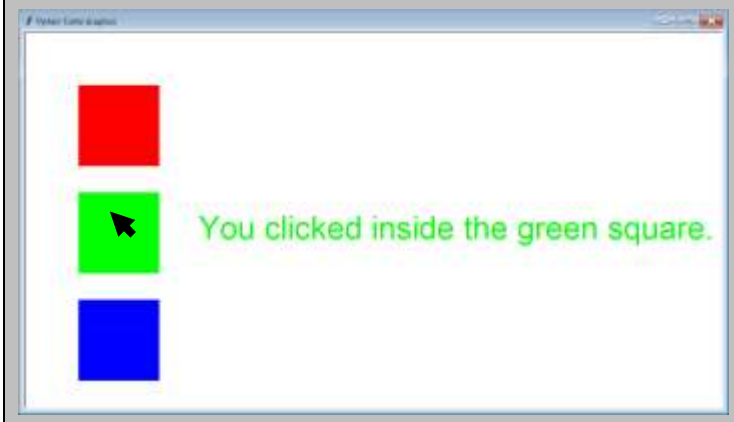
Program **MouseEvent10.py**, in Figure 18.11, displays three colored squares. The mission of the program is to see if the program can tell if the mouse is clicked anywhere in one of the three squares or anywhere outside the squares. This is done by using *compound conditions* and *ranges*. Look at the program in its entirety first with its execution and then more explanation will follow.

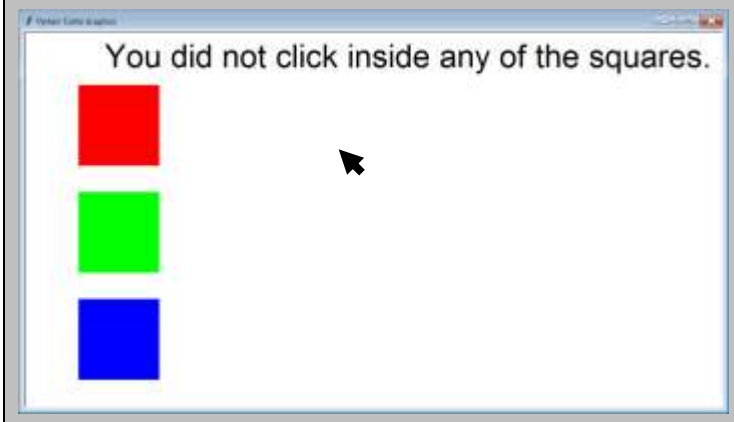
Figure 18.11

```
# MouseEvents10.py
# This program uses "ranges" to determine
# if the user clicked inside a certain
# rectangular (or square) area of the screen.

from Graphics import *

def displayBoxes():
    setColor("red")
    fillRectangle(100,100,250,250)
    setColor("green")
    fillRectangle(100,300,250,450)
    setColor("blue")
    fillRectangle(100,500,250,650)

def locate(x,y):
    clear()
    displayBoxes()
    x,y = traditionalXY(x,y)
    if 100 <= x <= 250 and 100 <= y <= 250:
        setColor("red")
        drawString("You clicked inside the red square.",325,200,"Arial",36,"normal")
    elif 100 <= x <= 250 and 300 <= y <= 450:
        setColor("green")
        drawString("You clicked inside the green square.",325,400,"Arial",36,"normal")
    elif 100 <= x <= 250 and 500 <= y <= 650:
        setColor("blue")
        drawString("You clicked inside the blue square.",325,600,"Arial",36,"normal")
    else:
        setColor("black")
        drawString("You did not click inside any of the squares.",150,80,"Arial",36,"normal")


##########
#  MAIN  #
##########

beginGrfx(1300,700)
displayBoxes()
onscreenclick(locate)
endGrfx()

```

Initial Output:



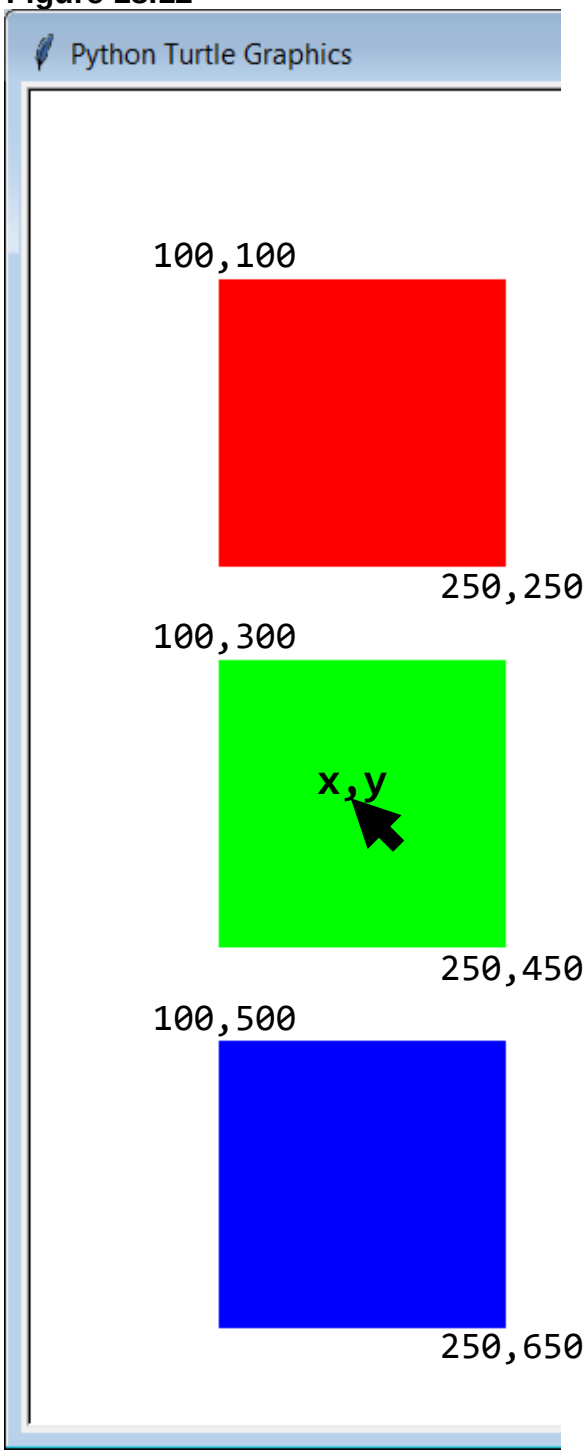




OK, so how does this work. To determine if the user clicked inside one of the squares, we take the **x** and **y** value of the clicked coordinate and compare them to the coordinates of the top-left corner and bottom-right corner of the square.

Figure 18.12 shows an enlarged depiction of the three squares from this program's output. The coordinates of each of the squares is super-imposed on the image.

Figure 18.12



Let us look assume the user just clicked in the green square. The **locate** method from this program receives the **x** and **y** value of the exact click coordinate from **onscreenclick**.

To confirm that this click occurred *inside* the green square, we need to see that 2 specific things are true. First, we need to see that the **x** value is between **100** and **250**. These are the **x** values of the upper-left corner and lower-right corner of the square respectively. This is why the **elif** command on line 27 says:

```
100 <= x <= 250
```

Second, we need to see that the **y** value is between **300** and **450**. These are the **y** values of the upper-left corner and lower-right corner of the square respectively. This is why the **elif** command on line 27 also says:

```
and 300 <= y <= 450
```

If both of these conditions are true, we know the click occurred inside the green rectangle.

The previous program may have been a bit tedious, but its purpose was to help you understand the process of what goes on when trying to determine if you clicked in a certain area. Now that we understand this process, we are going to do things in a simpler way. The **Graphics** library has a special function called

inside which does everything in the compound condition of ranges from the previous program, and even does the coordinate conversion between *Turtle Graphics* and *Traditional Graphics* as well. Program **MouseEvents11.java**, in Figure 18.13, does the exact same thing as the previous program, but now that it takes advantage of the **inside** function, the code is less cluttered. The output is identical to the previous program.

Figure 18.13

```
# MouseEvents11.py
# This program does the exact same thing as the previous program.
# The code is now a little more efficient because it uses the <inside>
# function of the <Graphics> library.

from Graphics import *

def displayBoxes():
    setColor("red")
    fillRectangle(100,100,250,250)
    setColor("green")
    fillRectangle(100,300,250,450)
    setColor("blue")
    fillRectangle(100,500,250,650)

def locate(x,y):
    clear()
    displayBoxes()
    if inside(x,y,100,100,250,250):
        setColor("red")
        drawString("You clicked inside the red square.",325,200,"Arial",36,"normal")
    elif inside(x,y,100,300,250,450):
        setColor("green")
        drawString("You clicked inside the green square.",325,400,"Arial",36,"normal")
    elif inside(x,y,100,500,250,650):
        setColor("blue")
        drawString("You clicked inside the blue square.",325,600,"Arial",36,"normal")
    else:
        setColor("black")
        drawString("You did not click inside any of the squares.",150,80,"Arial",36,"normal")

```

```
32
33 ##########
34 #  MAIN  #
35 ##########
36
37 beginGrfx(1300,700)
38 displayBoxes()
39 onscreenclick(locate)
40 endGrfx()
41
```

There is one more point that needs to be made in this section. In the past couple programs, 3 colored squares were drawn on the graphics window. Clicking on different squares caused the program to display different messages. Here is the thing, technically speaking, the squares are not necessary. Program **MouseEvents12.java**, in Figure 18.14, is almost identical to the previous program. The only difference is that the 2 calls to the **displayBoxes** procedure have been commented out.

When you run the program, all you see is a blank screen. Even though there are no solid color squares visible on the screen, the rectangular (or square) areas defined by the **inside** function are still there. These areas may be invisible, but they can still be clicked even though there is nothing visible on the screen. When you look at the output, you should see that the program still behave the same way, even though the squares are no longer visible.

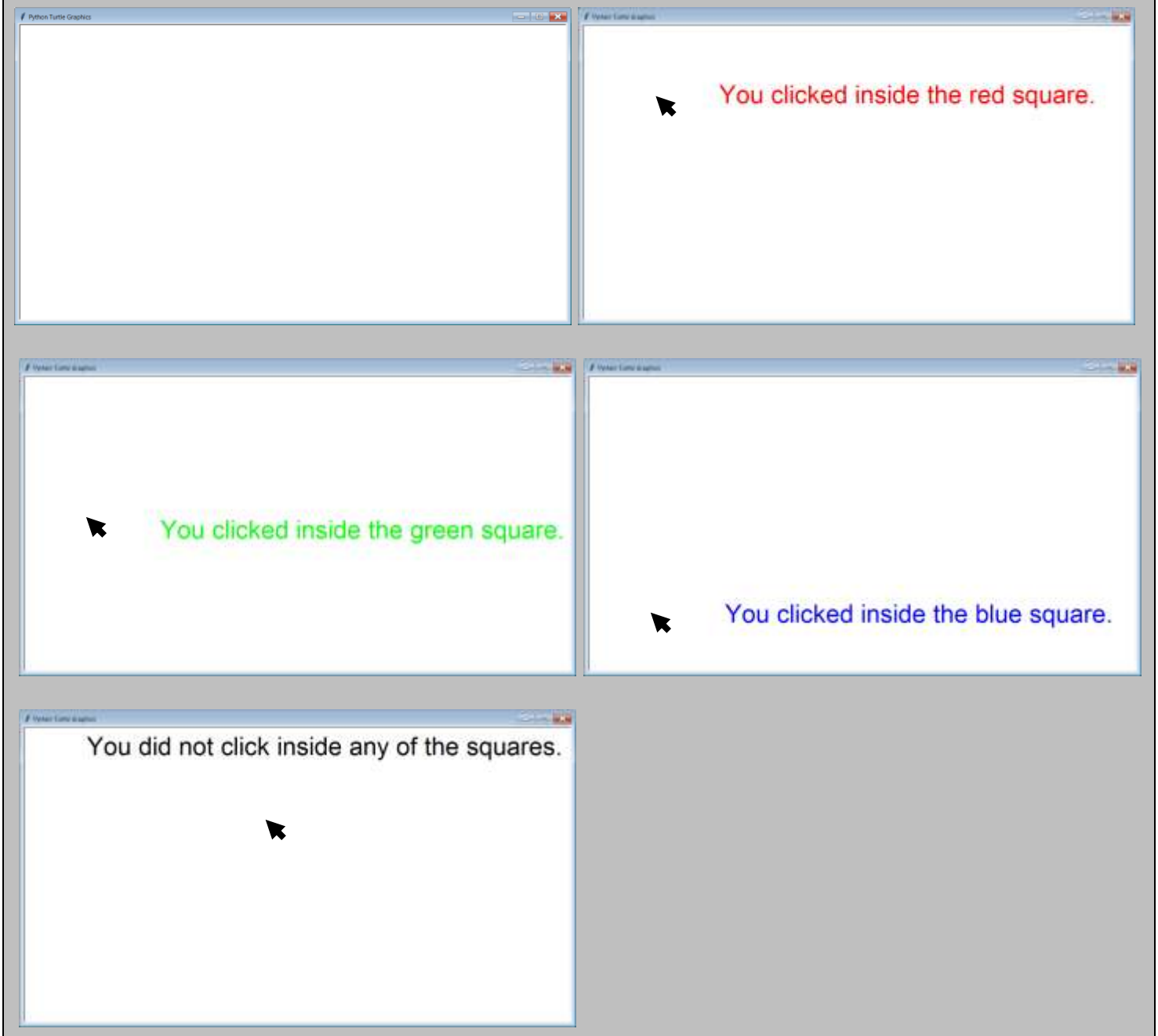
Do keep in mind that to expect a user to click on some empty area of the screen in order for something to happen is not user-friendly at all. If you expect the user to click somewhere, draw something there for them to click on.

Figure 18.14

```
1 # MouseEvents12.py
2 # The only difference between this program and the previous program is that
3 # in this one the calls to procedure <displayBoxes> have been commented out.
4 # Note that even though the boxes are not displayed, you can still click on
5 # them... or at least, you can click on the area they occupied.
6
7 from Graphics import *
```

```
 8
 9 def displayBoxes():
10     setColor("red")
11     fillRectangle(100,100,250,250)
12     setColor("green")
13     fillRectangle(100,300,250,450)
14     setColor("blue")
15     fillRectangle(100,500,250,650)
16
17 def locate(x,y):
18     clear()
19     #displayBoxes()
20     if inside(x,y,100,100,250,250):
21         setColor("red")
22         drawString("You clicked inside the red square.",325,200,"Arial",36,"normal")
23     elif inside(x,y,100,300,250,450):
24         setColor("green")
25         drawString("You clicked inside the green square.",325,400,"Arial",36,"normal")
26     elif inside(x,y,100,500,250,650):
27         setColor("blue")
28         drawString("You clicked inside the blue square.",325,600,"Arial",36,"normal")
29     else:
30         setColor("black")
31         drawString("You did not click inside any of the squares.",150,80,"Arial",36,"normal")
32
33
34 ##########
35 #  MAIN  #
36 ##########
37
38 beginGrfx(1300,700)
39 #displayBoxes()
40 onscreenclick(locate)
41 endGrfx()
```

Initial Output:



There are actually <u>many</u> more mouse events that are available in Python, but they require code that is far more complicated than **onscreenclick**. These will not be covered in this first-year class. For now, we are going to move on to *Key Events*.

18.5 Key Events

Mouse Events are not the only kind of events that can be *triggered* in Python. There are also *Key Events*. Simply pressing any of the keys on the keyboard triggers an event. You may wonder if this is the same thing as *Keyboard Input*. It is not. With keyboard input, the entire program stops and waits for you to enter something from the keyboard. The program does not resume until you enter something and press the <enter> key. *Key Events* are different. With Key Events, while the program is executing, it *listens* for any key activity. If the user presses a specific key, then a specific procedure is called.

Detecting Normal Keys

Program **KeyEvents01.py**, in Figure 18.15, demonstrates the **onkey** command. This works in a manner similar to **onscreenclick** from the previous section. Let us compare the 2 commands below:

```
onscreenclick(displayDot,btn=1)
onkey(displayq,"q")
```

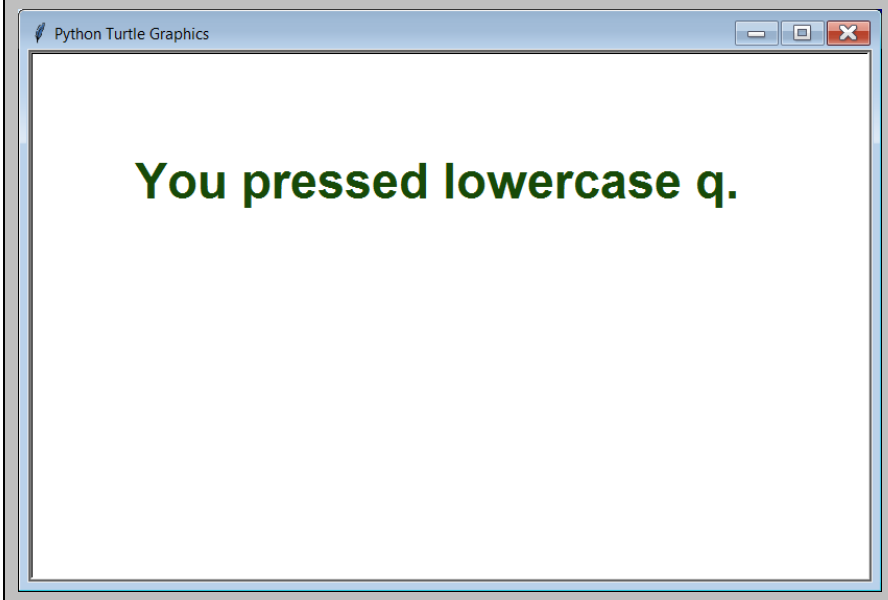
We see that the first command, **onscreenclick**, will call the **displayDot** procedure if "button 1" (the left one) is clicked. The second command, **onkey**, is very similar. This will call the **displayq** procedure if lowercase '**q**' is "keyed".

When you first run this program, the screen is blank. Anytime you press lowercase '**q**', the message "**You pressed lowercase q.**" is displayed in a random color. If you press any other key, even CAPITAL '**Q**', nothing happens.

Figure 18.15

```
1 # KeyEvents01.py
2 # This program introduces "Key Interactivity"
3 # by displaying a randomly colored message
4 # anytime lowercase 'q' is typed.
5 # NOTE: This not the same as "Keyboard Input".
6 # ALSO: The <listen> command is necessary for
7 # the <onkey> command to function properly.
8
```

```
 9
10 from Graphics import *
11
12
13 def displayq():
14     clear()
15     setRandomColor()
16     drawString("You pressed lowercase q.",100,150,
"Arial",28,"bold")
17     update()
18
19
20
21 ##########
22 #  MAIN  #
23 ##########
24
25
26 beginGrfx(800,500)
27 listen()
28 onkey(displayq,"q")
29 endGrfx()
```

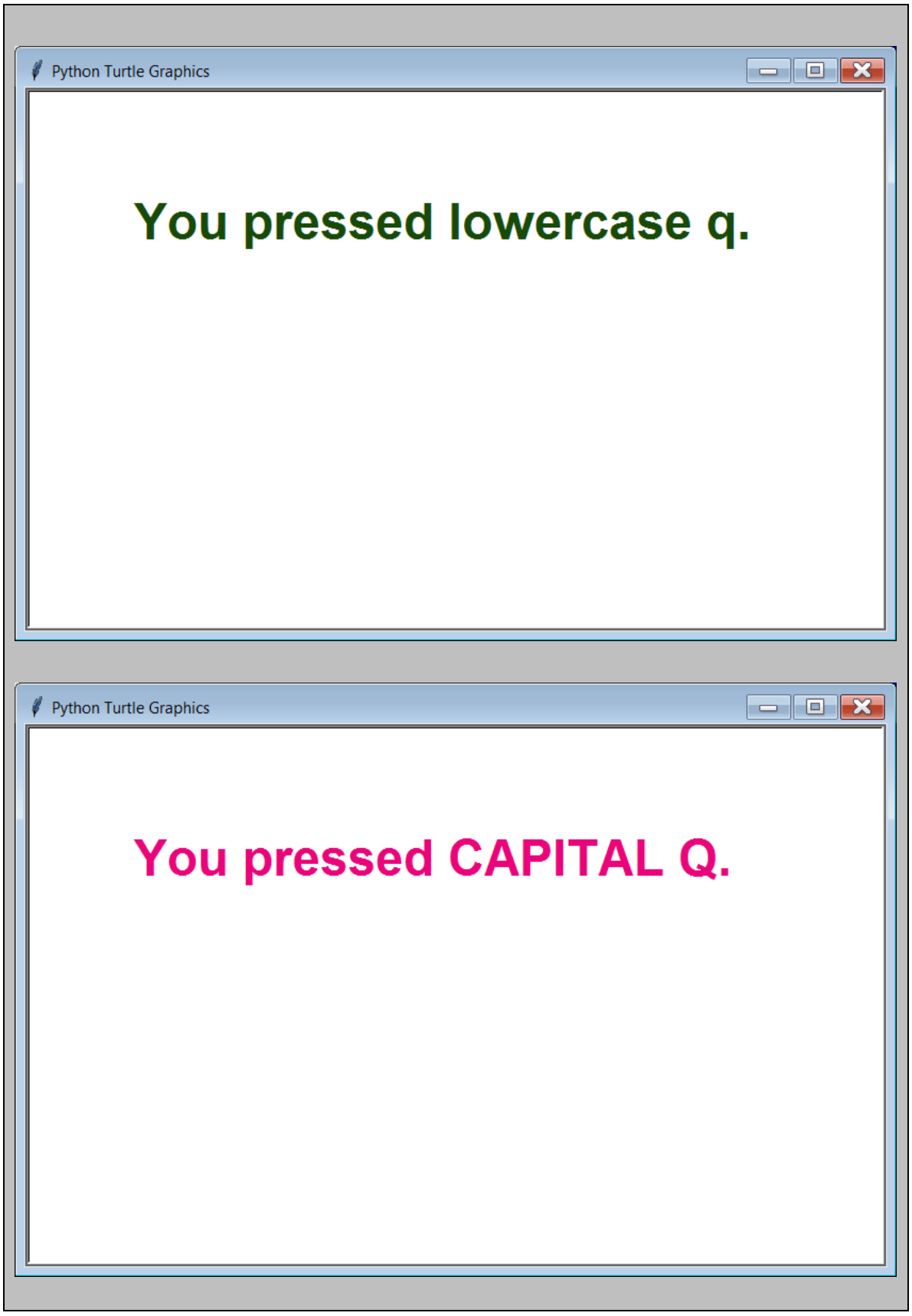


One thing that you may not have noticed is the **listen** command. Remember that I said, "…while the program is executing, it *listens* for any key activity." The **listen** command is necessary to make this happen. Without it, **onkey** will not work.

Program **KeyEvents02.py**, in Figure 18.16, demonstrates how to "listen" for multiple different keys with 2 separate **onkey** commands. It also shows that CAPITAL '**Q**' and lowercase '**q**' are treated as different keys.

Figure 18.16

```
# KeyEvents02.py
# This program demonstrates that the computer
# can "listen" for multiple different keys.
# It also shows that CAPITAL 'Q' and lowercase 'q'
# are treated as different keys.


from Graphics import *


def displayq():
    clear()
    setRandomColor()
    drawString("You pressed lowercase q.",100,150,"Arial",28,"bold")
    update()

def displayQ():
    clear()
    setRandomColor()
    drawString("You pressed CAPITAL Q.",100,150,"Arial",28,"bold")
    update()



##########
#  MAIN  #
##########


beginGrfx(800,500)
listen()
onkey(displayq,"q")
onkey(displayQ,"Q")
endGrfx()
```


Python Turtle Graphics
You pressed lowercase q.
Python Turtle Graphics
You pressed CAPITAL Q.

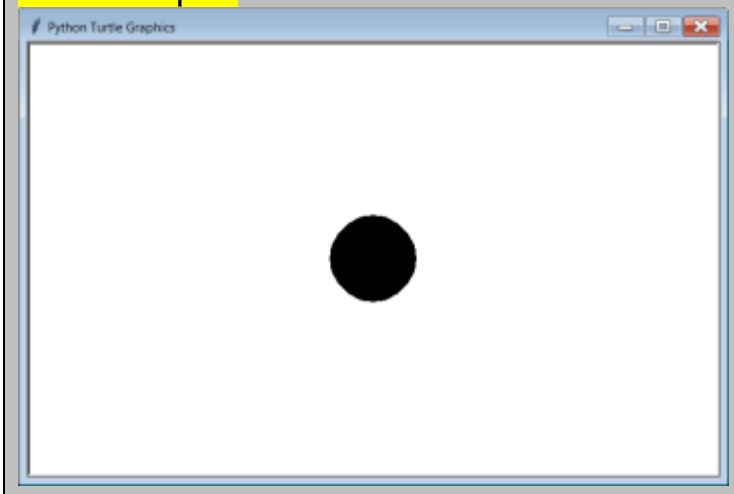
Program **KeyEvents03.py**, in Figure 18.17, is a bit more advanced that the previous 2 programs. When you look at the output, all you see is a circle. Now press the letters '**w**', '**a**', '**s**', or '**z**' and watch what happens. You will see that you can move the circle up, down, left or right. The way this works is the program stores the **x** and **y** values of the center of the circle. If you press the letter '**w**' (which is used for up) then **10** is subtracted from the **y** value. Pressing '**z**' (which is used for down) will add **10** to the **y** value. Pressing '**a**' (which is used for left) will subtract **10** from the **x** value and pressing '**s**' (which is used for right) will add **10** to the **x** value.

Figure 18.17

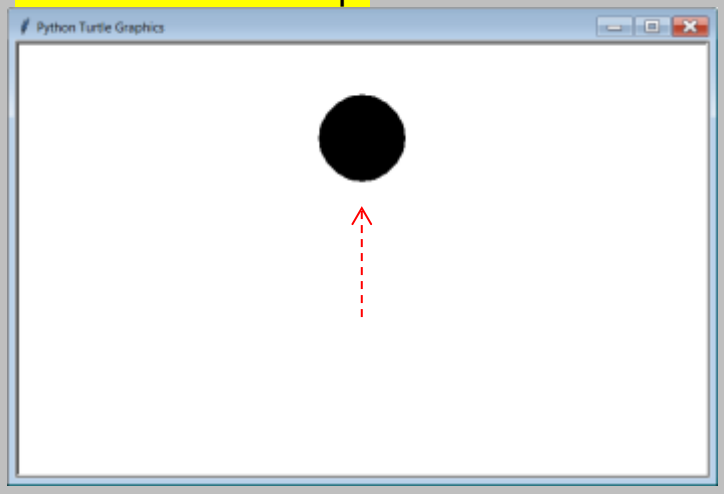
```
 1 # KeyEvents03.py
 2 # This program controls the movement of a
 3 # circle on the screen using the letters:
 4 #    w
 5 #   a s
 6 #    z
 7
 8
 9 from Graphics import *
10
11
12 def moveUp():
13     global y
14     y -= 10
15     clear()
16     fillCircle(x,y,50)
17     update()
18
19 def moveDown():
20     global y
21     y += 10
22     clear()
23     fillCircle(x,y,50)
24     update()
25
```

```
26 def moveLeft():
27     global x
28     x -= 10
29     clear()
30     fillCircle(x,y,50)
31     update()
32 
33 def moveRight():
34     global x
35     x += 10
36     clear()
37     fillCircle(x,y,50)
38     update()
39 
40 
41 
42 ##########
43 #  MAIN  #
44 ##########
45 
46 
47 x = 400
48 y = 250
49 
50 beginGrfx(800,500)
51 listen()
52 onkey(moveUp,"w")
53 onkey(moveDown,"z")
54 onkey(moveLeft,"a")
55 onkey(moveRight,"s")
56 fillCircle(x,y,50)
57 endGrfx()
```

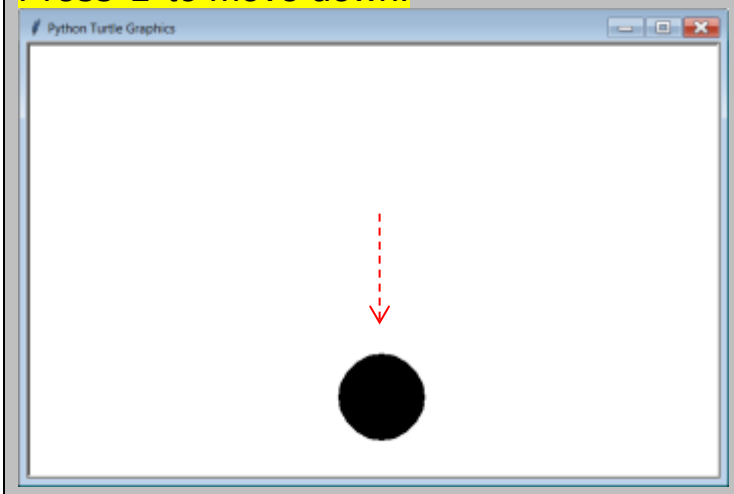
Initial Output:



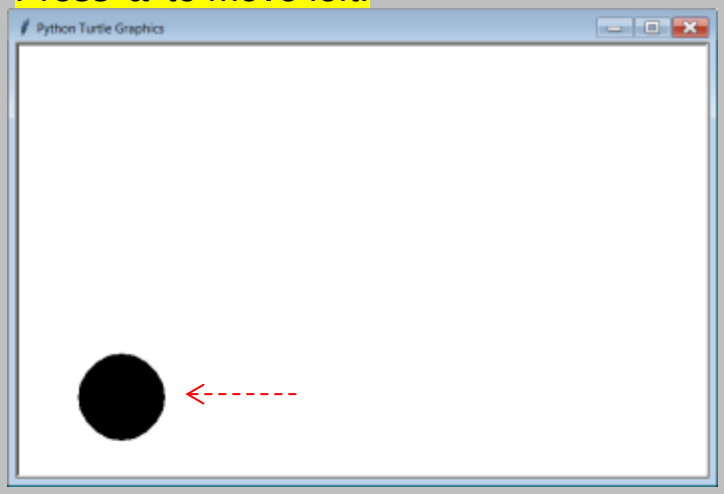
Press '**w**' to move up.



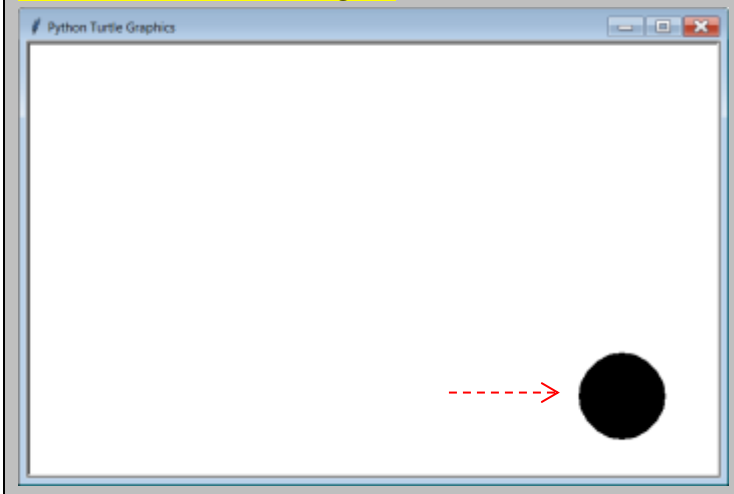
Press '**z**' to move down.



Press '**a**' to move left.

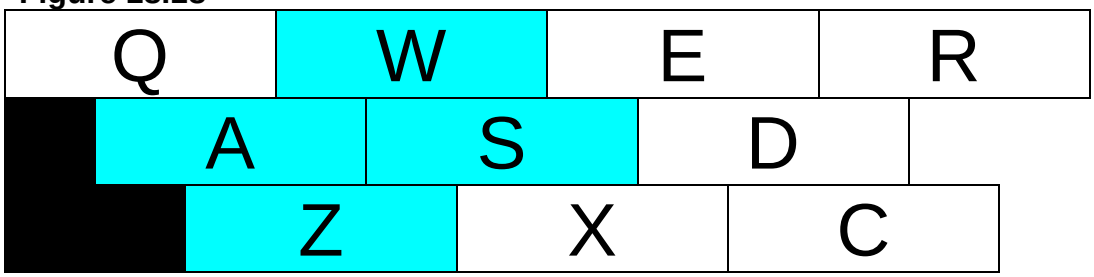


Press '**s**' to move right.



This is all fine. You may understand how the program works, but you might also be wondering why I chose such seemingly random letters like '**w**', '**a**', '**s**', and '**z**'. If you look at the left side of your keyboard, you should notice a pattern similar to what is shown in Figure 18.18:

Figure 18.18



Do you see how these 4 keys form something similar to *compass* directions? Many older computer video games would use these keys to control the game. You may be thinking, "What about the arrow keys?" The first computer keyboards did not have arrow keys. Today that is not an issue, and Python has a nice simple way to detect not only the arrow keys, but other keys as well.

Detecting Special Keys

Program **KeyEvents04.py**, in Figure 18.19, demonstrates how to detect "Special Keys" like <Insert>, <Delete>, <Home> and the arrow keys. It works very much in the same manner as detecting "Normal Keys". The only difference is that you need to know the "key symbol" for the special key that you are detecting. These key symbols are not exactly complex. They are meant to be intuitive. For example, the key symbol for the <Home> key is **"Home"** and the key symbol for the <Up Arrow> is **"Up"**.

This program will do everything that the previous program did, but now more intuitive arrow keys are used; however, this program also has a few extra features. If you press the <Insert> key the circle will get bigger. If you press the <Delete> key the circle will get smaller. Pressing the <Home> key will restore **x**, **y**, and **r** to their original values. This returns the circle to its original size and moves it back to its original location in the center of the graphics window.

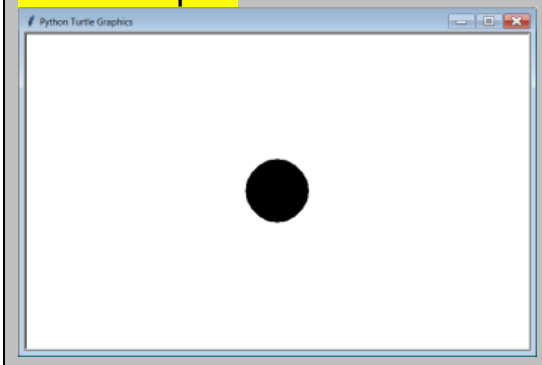
Figure 18.19

```
1 # KeyEvents04.py
2 # This program shows that <onkey> works with
3 # "Special Keys" as well like the arrow keys,
4 # as well as "Insert", "Delete" and "Home".
5
```

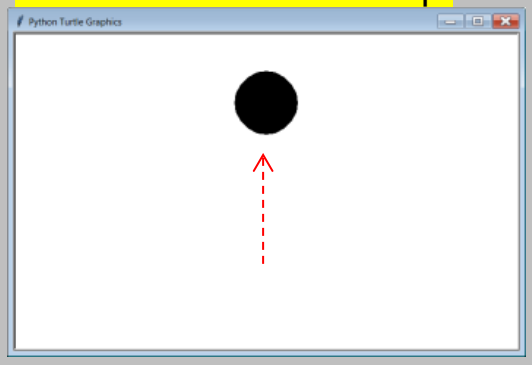
```
 6
 7 from Graphics import *
 8
 9
10 def moveUp():
11    global y
12    y -= 10
13    clear()
14    fillCircle(x,y,r)
15    update()
16
17 def moveDown():
18    global y
19    y += 10
20    clear()
21    fillCircle(x,y,r)
22    update()
23
24 def moveLeft():
25    global x
26    x -= 10
27    clear()
28    fillCircle(x,y,r)
29    update()
30
31 def moveRight():
32    global x
33    x += 10
34    clear()
35    fillCircle(x,y,r)
36    update()
37
38 def bigger():
39    global r
40    r += 10
41    clear()
42    fillCircle(x,y,r)
43    update()
44
45 def smaller():
46    global r
47    r -= 10
```

```
48    if r < 0:
49       r = 0
50    clear()
51    fillCircle(x,y,r)
52    update()
53
54 def center():
55    global x,y,r
56    x = 400
57    y = 250
58    r = 50
59    clear()
60    fillCircle(x,y,r)
61    update()
62
63
64
65 ##########
66 #  MAIN  #
67 ##########
68
69
70 x = 400
71 y = 250
72 r = 50
73
74 beginGrfx(800,500)
75 listen()
76 onkey(moveUp,"Up")
77 onkey(moveDown,"Down")
78 onkey(moveLeft,"Left")
79 onkey(moveRight,"Right")
80 onkey(bigger,"Insert")
81 onkey(smaller,"Delete")
82 onkey(center,"Home")
83 fillCircle(x,y,r)
84 endGrfx()
```

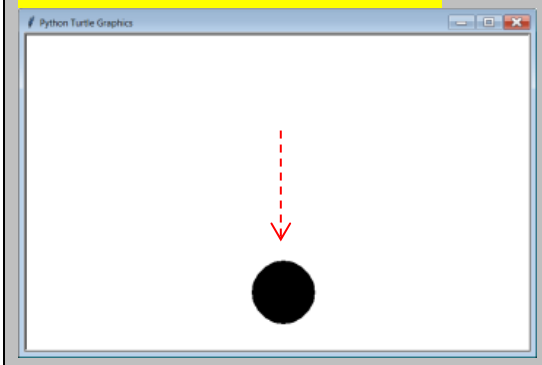
Initial Output:



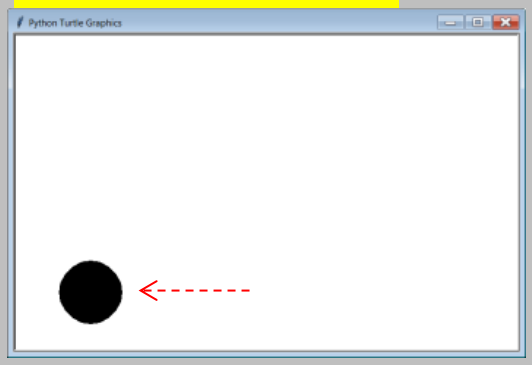
Press the <↑> to move up.



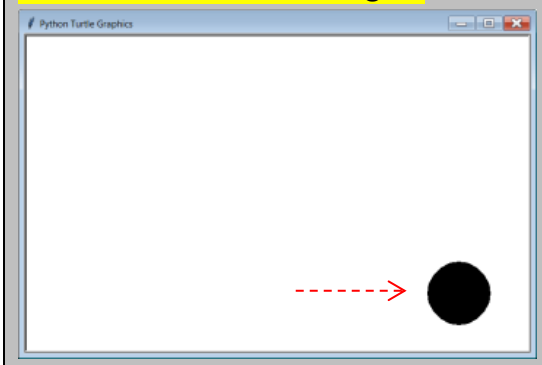
Press <↓> to move down.



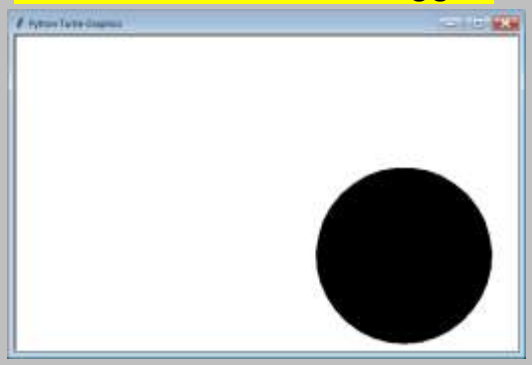
Press <←> to move left.



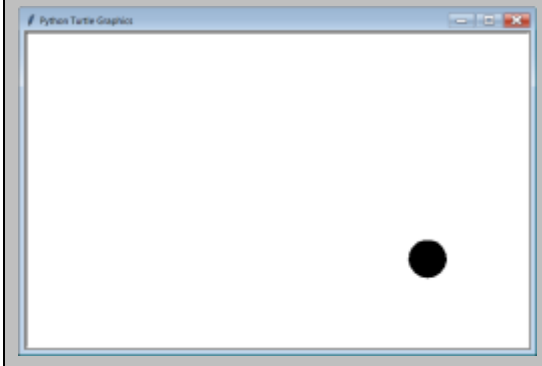
Press <→> to move right.



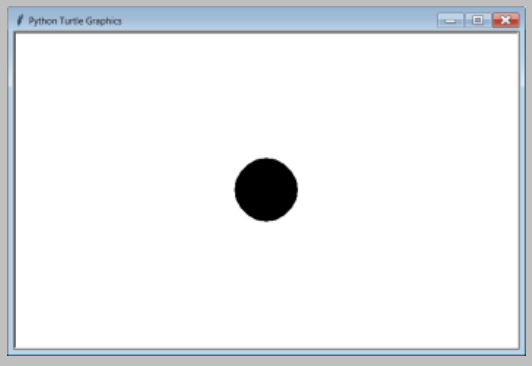
<Insert> makes circle bigger.



<Delete> makes circle smaller.



<Home> resets the screen.



One thing that has been a little tedious in the past couple programs is that moving the circle or changing its size requires you to keep pressing the same key repeatedly. It would be nicer if we could just hold the key down. The problem is that the **onkey** event only triggers once you type the key. To trigger it again, you need to release and then re-type the key. There is a better way. Python has another key event that is similar to **onkey**. This is **onkeypress**. Like **onkey**, **onkeypress** is triggered when you initially press the key; however, unlike **onkey**, **onkeypress** keeps triggering as long as you hold the key down.

Program **KeyEvents05.py**, in Figure 18.20, is almost identical to the previous program. The only difference is 6 of the 7 **onkey** commands have been replaced with **onkeypress**. The reason the last **onkey** was not changed to **onkeypress** is that it is detecting the <Home> key, which we are using the reset the screen. This is not something that we need to type repeatedly, so **onkey** is just fine.

Figure 18.20

```
 1 # KeyEvents05.py
 2 # This program replaces most of the <onkey>
 3 # commands with <onkeypress>.  This allows
 4 # the user to simply hold the key down rather
 5 # than having to type it repeatedly.
 6

   :    :    :    :    :    :    :    :

74
75 beginGrfx(800,500)
76 listen()
77 onkeypress(moveUp,"Up")
78 onkeypress(moveDown,"Down")
79 onkeypress(moveLeft,"Left")
80 onkeypress(moveRight,"Right")
81 onkeypress(bigger,"Insert")
82 onkeypress(smaller,"Delete")
83 onkey(center,"Home")
84 fillCircle(x,y,r)
85 endGrfx()
86
```

Program **KeyEvents06.py**, in Figure 18.21, demonstrates something important about the special string literal key symbols that are used with **onkey** and **onkeypress**. Specifically, they are case sensitive. Most of these string literals start with a CAPITAL letter and the rest is lowercase. If we type them differently, the program crashes and displays a long, complicated error message like the one shown below. The only part of the error message that seems to make any sense is the message: **bad event type or keysym "up"**. This basically says that "**up**" is a bad key symbol.

Figure 18.21

```
 1 # KeyEvents06.py
 2 # This program demonstrates that the string literal
 3 # "key symbols" used for the "Special Keys" are
 4 # "Case-Sensitive".
    :     :     :     :     :     :     :     :
74 beginGrfx(800,500)
75 listen()
76 onkeypress(moveUp,"up")  # should be "Up"
77 onkeypress(moveDown,"Down")
78 onkeypress(moveLeft,"Left")
79 onkeypress(moveRight,"Right")
80 onkeypress(bigger,"Insert")
81 onkeypress(smaller,"Delete")
82 onkey(center,"Home")
83 fillCircle(x,y,r)
84 endGrfx()
```

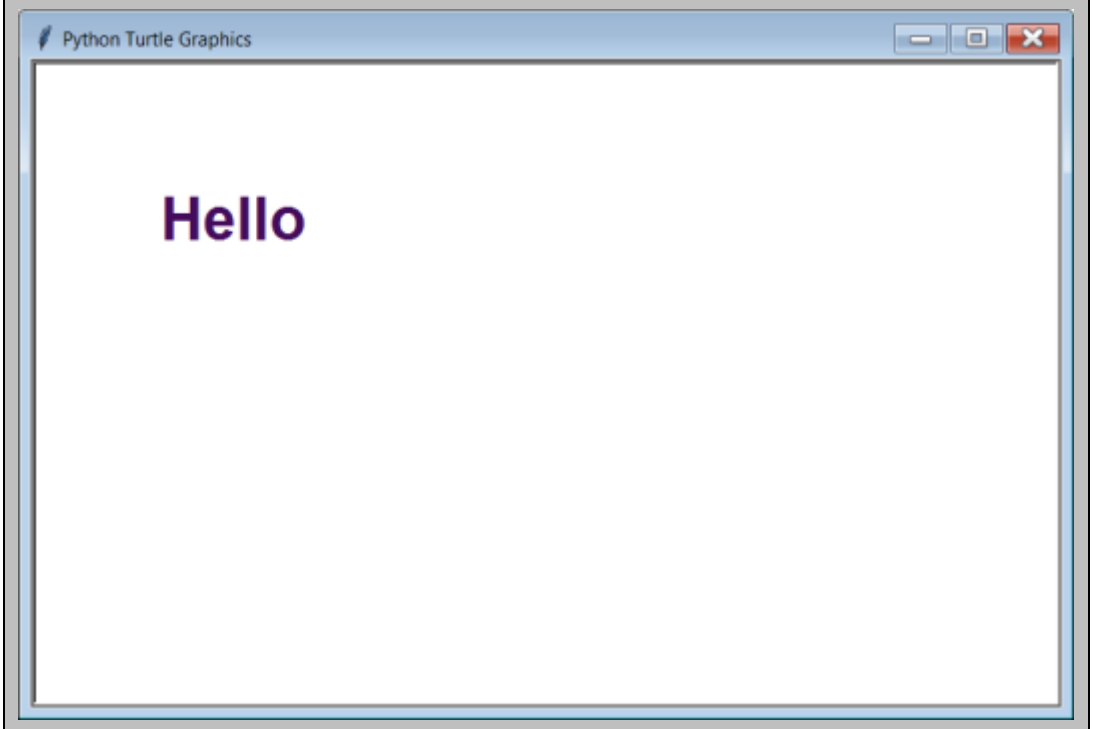
```
  ----jGRASP exec: python KeyEvents06.py
Traceback (most recent call last):
  File "KeyEvents06.py", line 76, in <module>
    onkeypress(moveUp,"up")
  File "<string>", line 8, in onkeypress
  File "turtle.py", line 1426, in onkeypress
    self._onkeypress(fun, key)
  File "turtle.py", line 705, in _onkeypress
    self.cv.bind("<KeyPress-%s>" % key, eventfun)
  File "turtle.py", line 416, in bind
    self._canvas.bind(*args, **kwargs)
  File "__init__.py", line 1245, in bind
    return self._bind(('bind', self._w), sequence, func, add)
  File "__init__.py", line 1200, in _bind
    self.tk.call(what + (sequence, cmd))
_tkinter.TclError: bad event type or keysym "up"

  ----jGRASP wedge2: exit code for process is 1.
  ----jGRASP: operation complete.
```

Program **KeyEvents07.py**, in Figure 18.22, attempts to display "**Hello**" anytime the user presses the <space bar>. Based on what we have seen so far, they seems like it would be simple. We just need to use **onkey** or **onkeypress** and make sure to use " " (a *space*) for the second parameter. Sounds simple. When you run the program, you may get the idea that the program works. Every time you press the <space bar>, "**Hello**" is displayed in a random color. Now see what happens when you type other keys. It still displays "**Hello**". The problem with using " " is that it does not just match the <space bar>, it matches any key that you press.

Figure 18.22

```
 1 # KeyEvents07.py
 2 # This program attempts to use the "space bar"
 3 # to make the computer display "Hello".  It may
 4 # seem to work at first, until you realize that
 5 # typing ANY key will make the computer display
 6 # "Hello".
 7
 8
 9 from Graphics import *
10
11
12 def display():
13     clear()
14     setRandomColor()
15     drawString("Hello",100,150,"Arial",28,"bold")
16     update()
17
18
19
20 ###########
21 #  MAIN   #
22 ###########
23
24
25 beginGrfx(800,500)
26 listen()
27 onkeypress(display," ")
28 endGrfx()
29
```

Output after ANY key is pressed:



Program **KeyEvents08.py**, in Figure 18.23, fixes the issue of the previous program. The secret is realizing that there is a special key symbol for the <space bar>. Instead of " ", you need to use **"space"**. Note that unlike the other string literal key symbols, **"space"** does not start with a capital letter.

Figure 18.23

```
 1 # KeyEvents07.py
 2 # This program corrects the issue of the previous
 3 # program by using the string literal key symbol
 4 # "space" instead of " ".
    :     :     :     :     :     :     :     :
23 beginGrfx(800,500)
24 listen()
25 onkeypress(display,"space")
26 endGrfx()
```

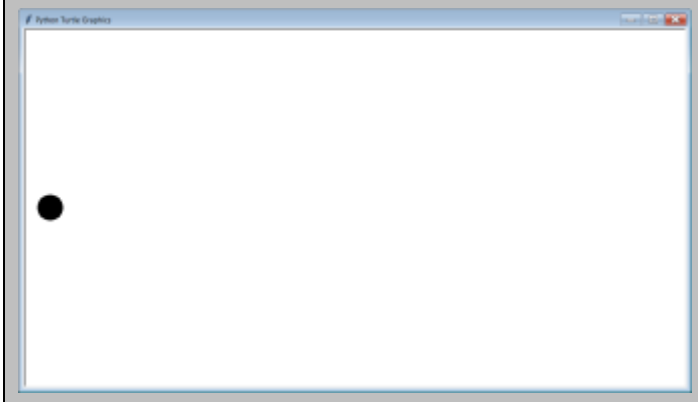
18.6 Computer Animation

Computer Animation makes many of today's modern movies possible. It also has been making video games possible for decades. The ability to move things on the screen makes your program much more dynamic and exciting. We are going to go through a series of 17 programs. The goal is to make a something move across the screen. We will not achieve this on the first program, but each program will get closer. We will start by attempting to move a circle across the screen. After we accomplish that, we will move on to more complicated images.

Do keep in mind that now, more than ever, you need to be executing these programs on your computer to see the output. It is impossible to show animated output in a textbook. I will be able to show the output of the first 3 programs since those do not have animation yet. Program **Animation01.py**, in Figure 18.24, is our first step in the animation journey.

Figure 18.24

```
# Animation01.py
# This program simply draws a solid black circle.
# This will eventually be "animated" as we go
# through the next few programs.


from Graphics import *


beginGrfx(1300,700)

fillCircle(50,350,25)

endGrfx()
```



You may be wondering why **Animation01.py** does nothing more than draw a circle. Here is why. Students get very excited when they do their final graphics project. The especially get excited when animation becomes involved. The following conversation occurs several times each year:

Eager Student: *Mr. Schram, my project is going to be so awesome! I am going to have this move that way and that move this way and this will be doing this and that will be doing that…*

Mr. Schram: *That sounds great. What do you have on the screen so far?*

Eager Student: *Nothing. I have not figured out animation yet.*

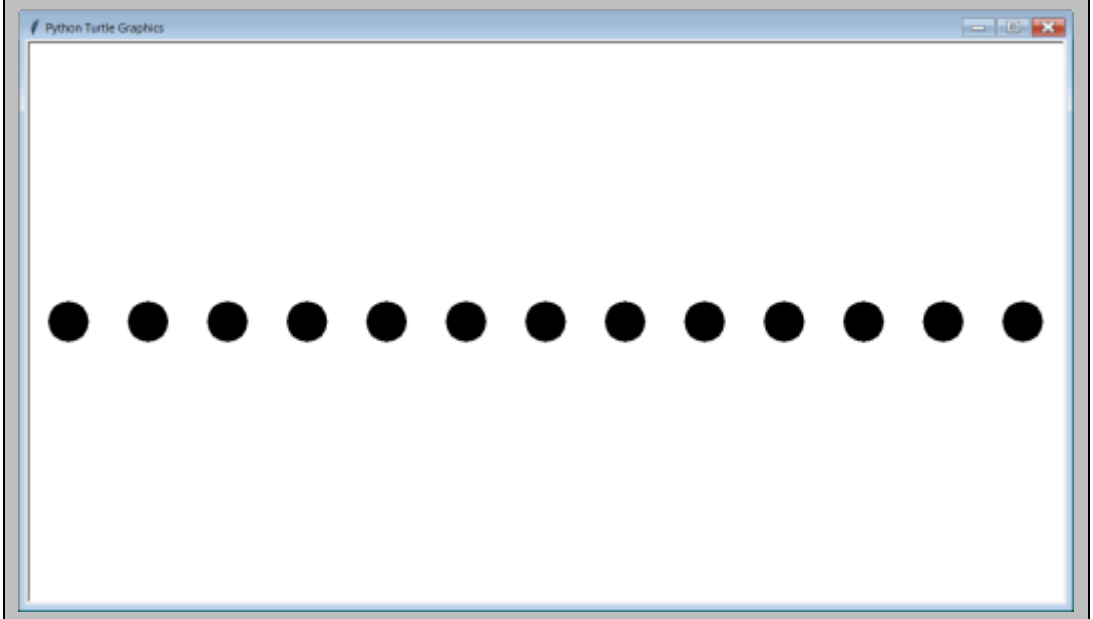
Here is my point. You cannot animate something if you do not have something to start with. If I want to animate a circle, I start by drawing a circle.

Program **Animation02.py**, in Figure 18.25, shows our next step. To make the circle look like it is moving across the screen, I need to draw it at different positions across the screen. I will draw the circle 13 times, once at the original location where the *x*-value is **50**, then again at the *x*-values of **150**, **250**, **350**, **450**, **550**, **650**, **750**, **850**, **950**, **1050**, **1150**, and **1250**.

Figure 18.25

```
 1 # Animation02.py
 2 # This program simply draws the solid black
 3 # circle at 13 different locations.
 4 # This is not yet animation because we see
 5 # all 13 circles at the same time.
 6
 7
 8 from Graphics import *
 9
10
11 beginGrfx(1300,700)
12
13 fillCircle(50,350,25)
14 fillCircle(150,350,25)
15 fillCircle(250,350,25)
16 fillCircle(350,350,25)
```

```
17 fillCircle(450,350,25)
18 fillCircle(550,350,25)
19 fillCircle(650,350,25)
20 fillCircle(750,350,25)
21 fillCircle(850,350,25)
22 fillCircle(950,350,25)
23 fillCircle(1050,350,25)
24 fillCircle(1150,350,25)
25 fillCircle(1250,350,25)
26
27 endGrfx()
28
```



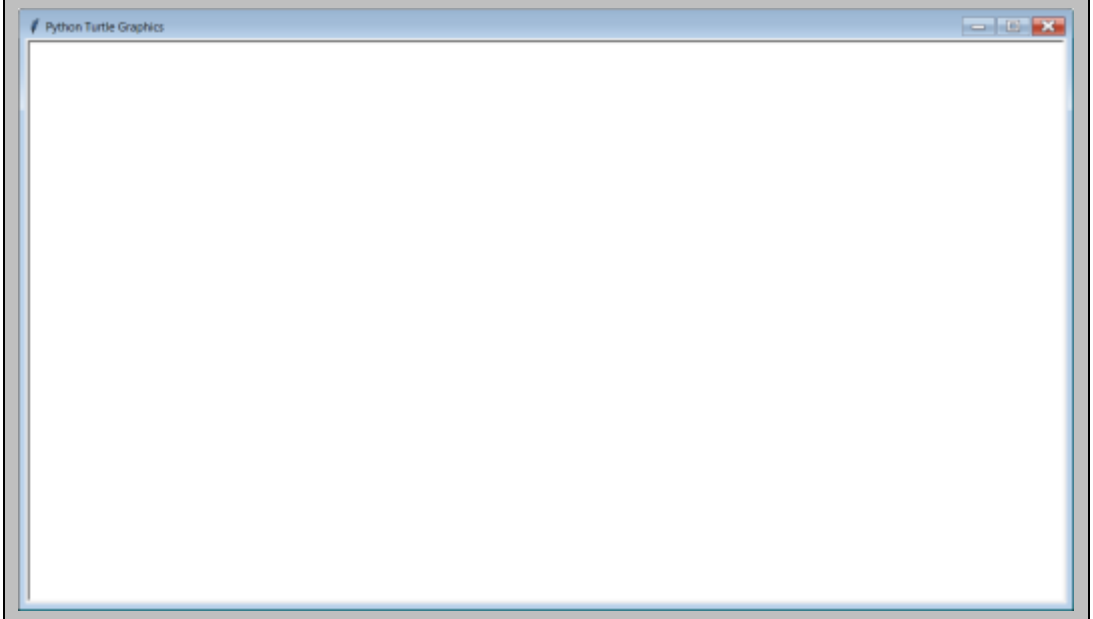
The output of program **Animation02.py** is hardly animation, but it shows something. There are 13 positions that I want my circle to take. The problem is I am seeing the circle at all 13 of those positions at the same time. Program **Animation03.py**, in Figure 18.26, attempts to deal with this problem by erasing each circle before the next circle is drawn. Erasing is accomplished by drawing a white circle on top of each black circle. Why white? The reason is **"white"** it is the background color.

Figure 18.26

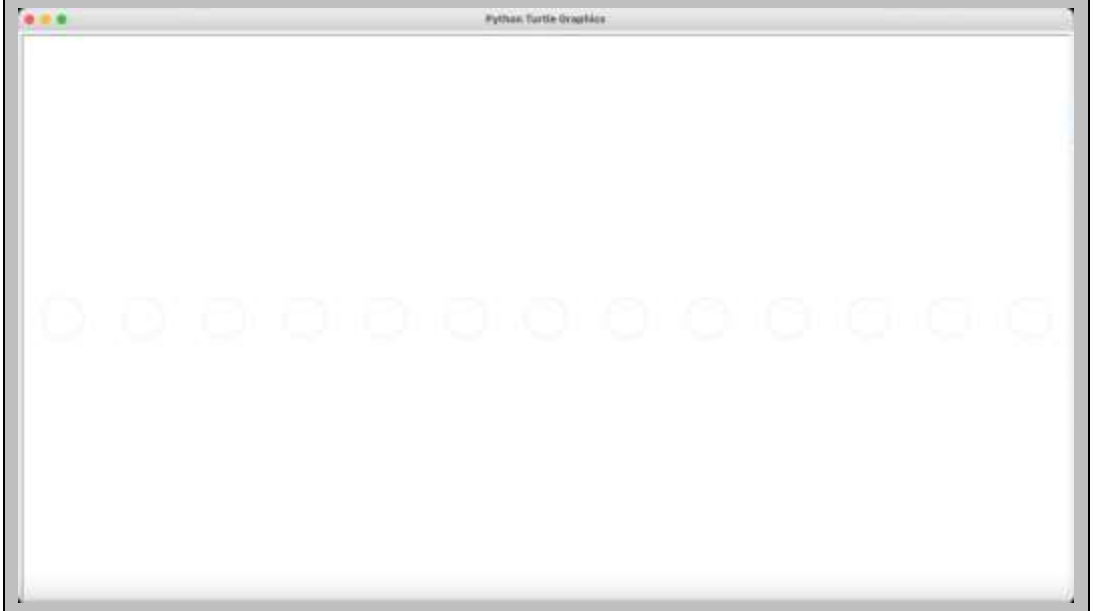
```
# Animation03.py
# This program erases every circle before it draws the
# next one.  This is necessary for animation; however,
# the process happens so fast that you cannot see the
# circle move.
# NOTE: On a Mac, you may see some circle outlines left
# behind because it did not completely erase the circles.


from Graphics import *


beginGrfx(1300,700)

setColor("black")
fillCircle(50,350,25)
setColor("white")
fillCircle(50,350,25)

setColor("black")
fillCircle(150,350,25)
setColor("white")
fillCircle(150,350,25)

setColor("black")
fillCircle(250,350,25)
setColor("white")
fillCircle(250,350,25)

setColor("black")
fillCircle(350,350,25)
setColor("white")
fillCircle(350,350,25)

setColor("black")
fillCircle(450,350,25)
setColor("white")
fillCircle(450,350,25)

setColor("black")
```

```
41 fillCircle(550,350,25)
42 setColor("white")
43 fillCircle(550,350,25)
44
45 setColor("black")
46 fillCircle(650,350,25)
47 setColor("white")
48 fillCircle(650,350,25)
49
50 setColor("black")
51 fillCircle(750,350,25)
52 setColor("white")
53 fillCircle(750,350,25)
54
55 setColor("black")
56 fillCircle(850,350,25)
57 setColor("white")
58 fillCircle(850,350,25)
59
60 setColor("black")
61 fillCircle(950,350,25)
62 setColor("white")
63 fillCircle(950,350,25)
64
65 setColor("black")
66 fillCircle(1050,350,25)
67 setColor("white")
68 fillCircle(1050,350,25)
69
70 setColor("black")
71 fillCircle(1150,350,25)
72 setColor("white")
73 fillCircle(1150,350,25)
74
75 setColor("black")
76 fillCircle(1250,350,25)
77 setColor("white")
78 fillCircle(1250,350,25)
79
80 endGrfx()
```

Windows Output:



Mac Output:



In the animation journey it looks like we have taken a giant step backwards. The output of program **Animation03.py** shows absolutely nothing on a Windows PC. On a Mac, it just shows the outlines of the 13 circles. It may not look like it, but the program did make the circle move across the screen. The problem is it moved so quickly, you did not get a chance to see it. Note that while I am using the word

"moved", the reality is that each individual circle was drawn and erased. On a Mac, the outlines were left behind because the circles did not completely erase.

Program **Animation04.py**, in Figure 18.27 cures both of these problems. The problem of the circle moving too fast is fixed by using the **delay** procedure. We need the image to stay on the screen long enough for your eyes to have a chance to see it. It does not need to be long, even a small fraction of a second is fine. The **delay** procedure has a single parameter which indicates the number of *milliseconds* that the computer should wait. A **delay** of **1000** would be *1 full second*. A **delay** of **2000** would be *2 full seconds*. A **delay** of **500** would be ½ a *second*.

The problem of the circle outlines that are left behind on a Mac is fixed by increasing the size of the radii on the erasing circles. Note that this is a band-aid fix for now. A better form of animation will be shown later that eliminates this problem completely.

This will be the first program that actually shows something resembling animation. Because of this, we cannot show you the output in the textbook. You will have to execute the remaining programs on your computer to see what is happening.

Figure 18.27

```
 1 # Animation04.py
 2 # This program adds a short 1 second delay
 3 # after each circle is drawn to give you a
 4 # chance to see it before it is erased.
 5 # NOTE: This is called "Draw-And-Erase Animation".
 6 # ALSO: The issue of the circle outlines on a Mac
 7 #       was fixed by making the radii of the
 8 #       erasing circles a little bigger.
 9
10
11 from Graphics import *
12
13
14 beginGrfx(1300,700)
15
16 setColor("black")
17 fillCircle(50,350,25)
18 delay(1000)
19 setColor("white")
20 fillCircle(50,350,27)
21
```

```
setColor("black")
fillCircle(150,350,25)
delay(1000)
setColor("white")
fillCircle(150,350,27)

setColor("black")
fillCircle(250,350,25)
delay(1000)
setColor("white")
fillCircle(250,350,27)

setColor("black")
fillCircle(350,350,25)
delay(1000)
setColor("white")
fillCircle(350,350,27)

setColor("black")
fillCircle(450,350,25)
delay(1000)
setColor("white")
fillCircle(450,350,27)

setColor("black")
fillCircle(550,350,25)
delay(1000)
setColor("white")
fillCircle(550,350,27)

setColor("black")
fillCircle(650,350,25)
delay(1000)
setColor("white")
fillCircle(650,350,27)

setColor("black")
fillCircle(750,350,25)
delay(1000)
setColor("white")
fillCircle(750,350,27)
```

```
63
64 setColor("black")
65 fillCircle(850,350,25)
66 delay(1000)
67 setColor("white")
68 fillCircle(850,350,27)
69
70 setColor("black")
71 fillCircle(950,350,25)
72 delay(1000)
73 setColor("white")
74 fillCircle(950,350,27)
75
76 setColor("black")
77 fillCircle(1050,350,25)
78 delay(1000)
79 setColor("white")
80 fillCircle(1050,350,27)
81
82 setColor("black")
83 fillCircle(1150,350,25)
84 delay(1000)
85 setColor("white")
86 fillCircle(1150,350,27)
87
88 setColor("black")
89 fillCircle(1250,350,25)
90 delay(1000)
91 setColor("white")
92 fillCircle(1250,350,27)
93
94 endGrfx()
95
```

Yes, we have achieved animation! It may not be earth-shattering animation, but the ball did move across the screen. Now, you probably noticed the program is very repetitive. In program **Animation04.py** there are 13 groups of 5-line code segments. The only thing that is different in each group is the *x*-value of the circle. In the first group this value is **50**. In the last group the value is **1250**. You should see that as you go from one group to the next the value increases by **100**. Now, think back. We have something repetitive that counts from **50** to **1250** by **100**. This is ideal for a **for** loop. Program **Animation05.py**, in Figure 18.28 does the exact same thing as program **Animation04.py**, but now a **for** loop is used and the program is much shorter.

Figure 18.28

```
 1 # Animation05.py
 2 # This program has the same output as the
 3 # previous program.  The program is now much
 4 # shorter because it uses a <for> loop.
 5
 6
 7 from Graphics import *
 8
 9
10 beginGrfx(1300,700)
11
12 for x in range(50,1251,100):
13     setColor("black")
14     fillCircle(x,350,25)
15     delay(1000)
16     setColor("white")
17     fillCircle(x,350,27)
18
19 endGrfx()
20
```

You have probably noticed the movement of the last 2 programs was a little choppy. It resembles the animation of a cheap cartoon. You may also have seen animated movies where the animation is much smoother and far superior. What makes the difference? Think of a film strip. There are 24 frames for every 1 second of film. When a live action movie is filmed, the video camera takes 24

distinct pictures per second. In a *made-for-the-screen* animated movie, the animators actually draw 24 distinct pictures for every 1 second of film. This is 129,600 pictures for a 90-minute movie. This is why animated movies take 5-7 *years* to produce.

Now think of a cheap 30-minute cartoon TV show. There is no way they are going to put in the same amount of effort when they have to mass produce as many as 65 episodes per season. For these cartoons, the animators only draw 4 distinct pictures for every 1 second of film. Each picture is duplicated on 6 frames in a row. This makes the animation look far less smooth.

Program **Animation06.py**, in Figure 18.29 improves the quality of the animation by increasing the number of circles drawn. Instead of the circle moving **100** pixels at a time, it will only move **10.** The delay is also changed from **1000** to **100** to compensate. We are now drawing 10 times as many circles, each of which is on the screen for 1/10 the amount of time as before. Execute the program and you should see smoother animation.

Figure 18.29

```
 1 # Animation06.py
 2 # This program makes the animation smoother
 3 # by using a smaller delay and a smaller
 4 # increment in the <for> loop.
 5
 6
 7 from Graphics import *
 8
 9
10 beginGrfx(1300,700)
11
12 for x in range(50,1251,10):
13     setColor("black")
14     fillCircle(x,350,25)
15     delay(100)
16     setColor("white")
17     fillCircle(x,350,27)
18
19 endGrfx()
20
```

Now you might ask, "*If changing the increment from 100 to 10 made the animation look better, would changing it from 10 to 1 make the animation look even better than that?*" Execute program **Animation07.py**, in Figure 18.30 and see for yourself. Note, the **delay** is reduced again to compensate.

Figure 18.30

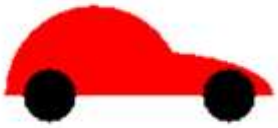
```
# Animation07.py
# This program makes the animation as smooth
# as possible by having an increment of just
# 1 pixel in the <for> loop.
# The delay is also made smaller.


from Graphics import *


beginGrfx(1300,700)

for x in range(50,1251,1):
    setColor("black")
    fillCircle(x,350,25)
    delay(10)
    setColor("white")
    fillCircle(x,350,27)

endGrfx()

```

The answer is "*Yes*." We now have the circle moving as smooth as possible.

Animating Complicated Images

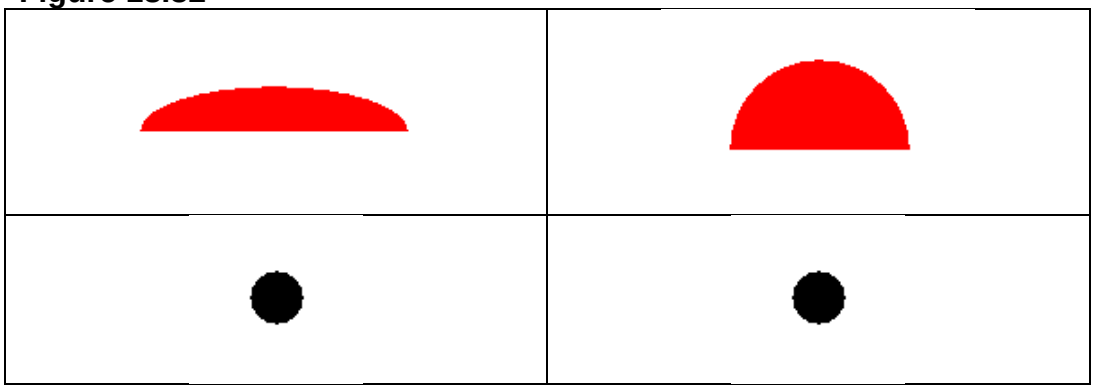
Suppose I want to animate something more than just a *circle*. What if I want to create the illusion of a *car* driving across the screen? Figure 18.31 shows the car that I wish to animate.

Figure 18.31



It should be understood that a car like this is not a *simple shape* that can be drawn with a single command. It is an overlapping composition of 4 filled shapes, specifically the 2 arcs and 2 circles shown in Figure 18.32.

Figure 18.32

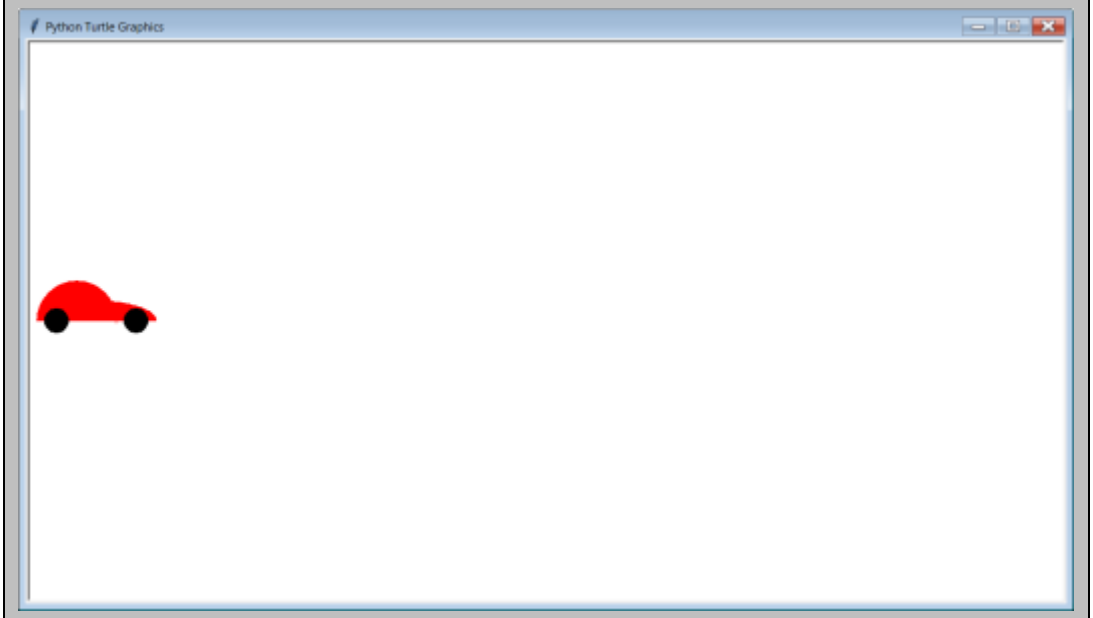


Program **Animation08.py**, in Figure 18.33, does not demonstrate any animation. It just draws the car shown in Figure 18.31. The next few programs will deal with making this more complicated image move across the screen.

Figure 18.33

```
 1 # Animation08.py
 2 # This program draws the car that will
 3 # be animated in the next few programs.
 4 # The car is an overlapping composition
 5 # of 4 filled shapes, specifically the
 6 # 2 arcs and 2 circles.
 7
 8
 9 from Graphics import *
10
11
```

```
12 beginGrfx(1300,700)
13
14 setColor("red")
15 fillArc(85,350,75,25,270,90)  # body
16 fillArc(60,350,50,50,270,90)  # hood
17 setColor("black")
18 fillCircle(35,350,15)         # rear wheel
19 fillCircle(135,350,15)        # front wheel
20
21 endGrfx()
22
```

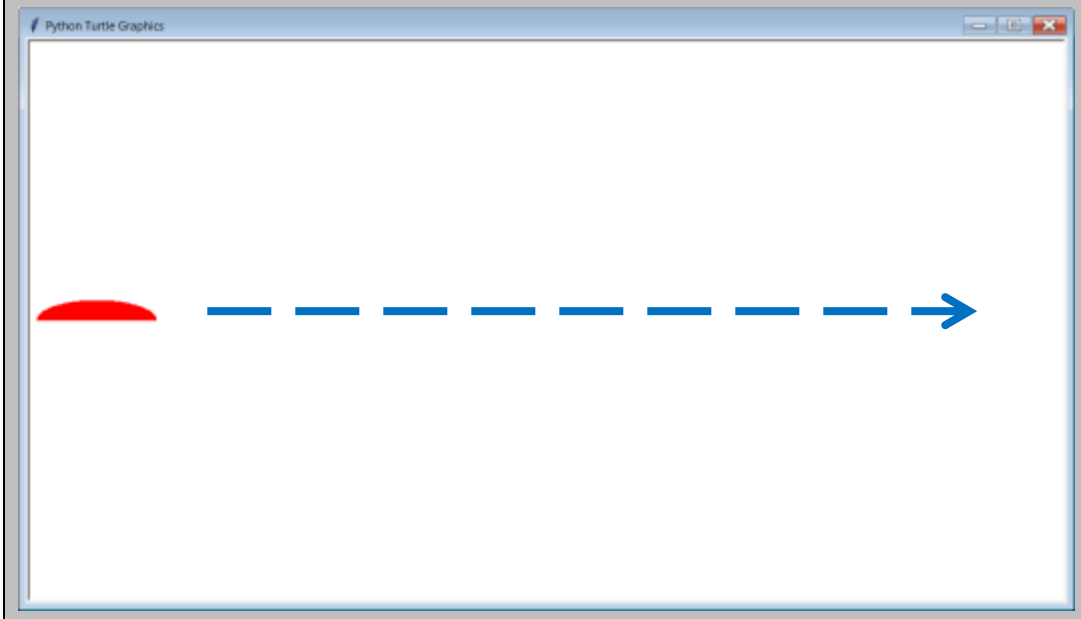


Program **Animation09.py**, in Figure 18.34, attempts to move the car across the screen. The reason I say "attempts" is the program does not work. It demonstrates a very common logic error among students trying to animate complicate images. Students remember the earlier program examples that moved a circle across the screen, so they do the same thing for the car. The problem is, they use a separate for loop for each shape in the car. The result is each *piece* of the car moves across the screen one at a time instead of all together.

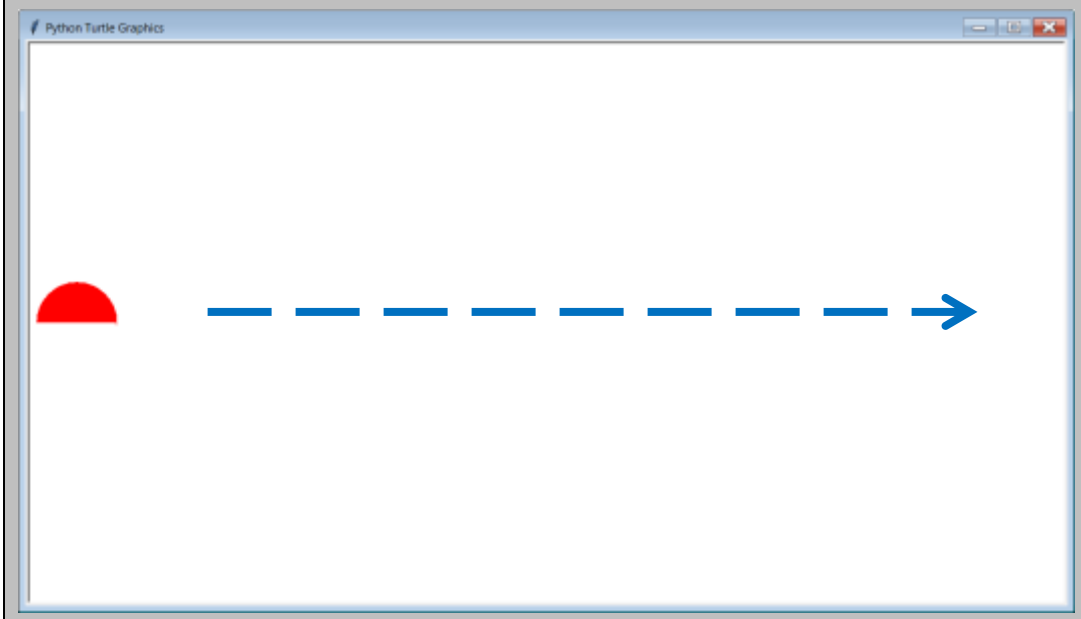
Figure 18.34

```
# Animation09.py
# This program demonstrates a common logic error
# when students try to move complicated images
# across the screen. The problem is each part of
# the car is in its own separate loop.  This means
# each part of the car moves across the screen one
# at a time instead of all together.  On top of this,
# on a Mac each piece may leave behind a trail.


from Graphics import *


beginGrfx(1300,700)

for x in range(85,1226,10):         # body
    setColor("red")
    fillArc(x,350,75,25,270,90)
    delay(50)
    setColor("white")
    fillArc(x,350,75,25,270,90)

for x in range(60,1201,10):         # hood
    setColor("red")
    fillArc(x,350,50,50,270,90)
    delay(50)
    setColor("white")
    fillArc(x,350,50,50,270,90)

for x in range(35,1176,10):         # rear wheel
    setColor("black")
    fillCircle(x,350,15)
    delay(50)
    setColor("white")
    fillCircle(x,350,15)

for x in range(135,1276,10):        # front wheel
    setColor("black")
    fillCircle(x,350,15)
    delay(50)
    setColor("white")
    fillCircle(x,350,15)

endGrfx()
```

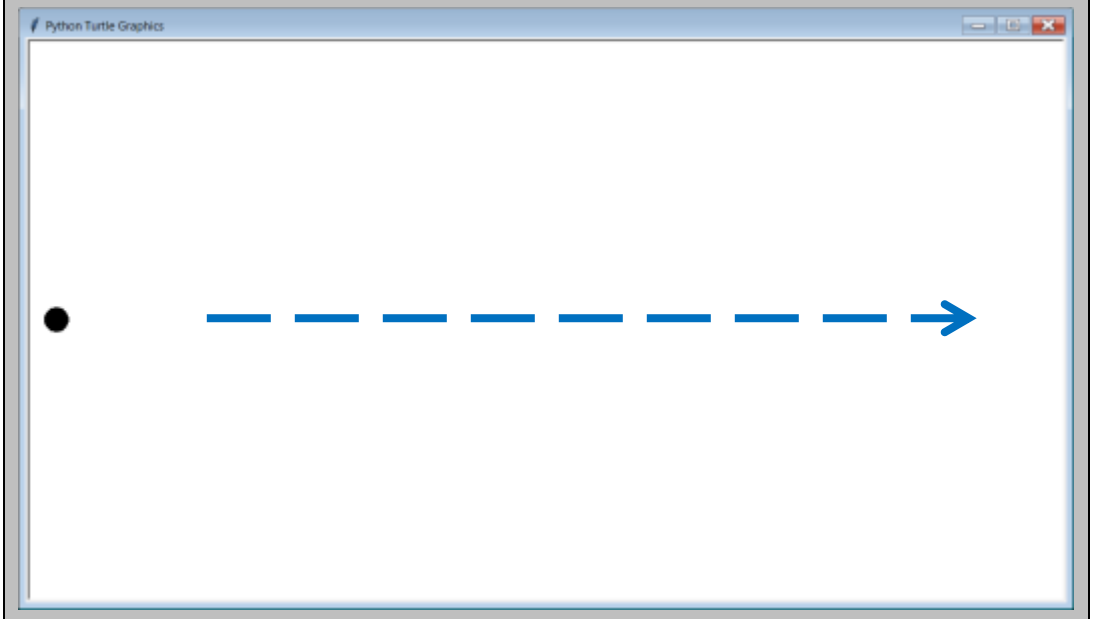
First, the car body moves across the screen.



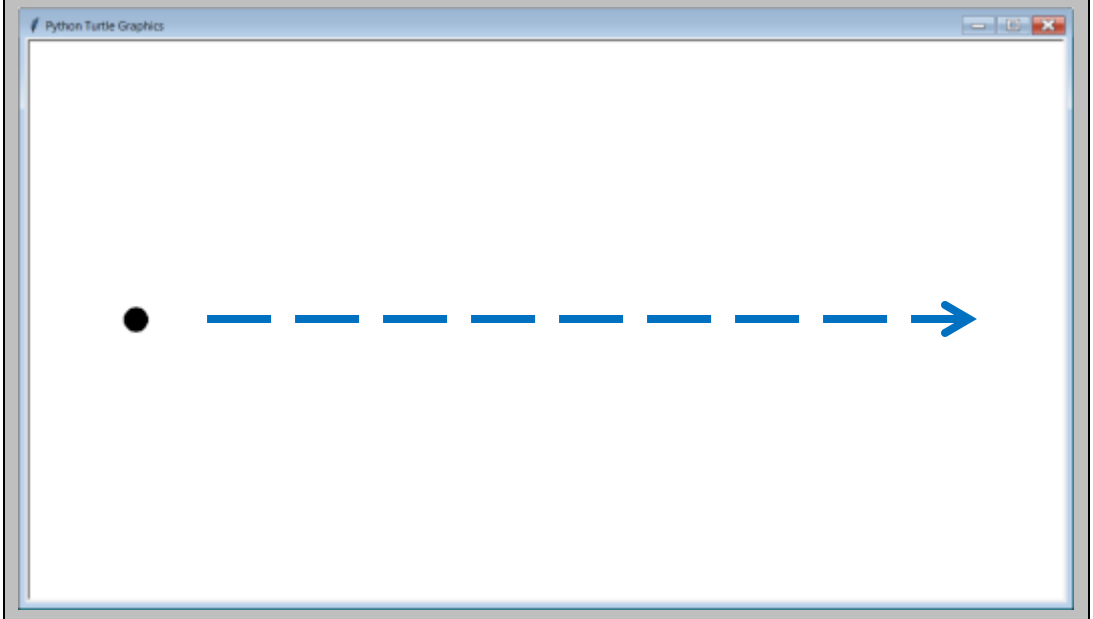
Then, the hood moves across the screen.



Then, the rear wheel moves across the screen.



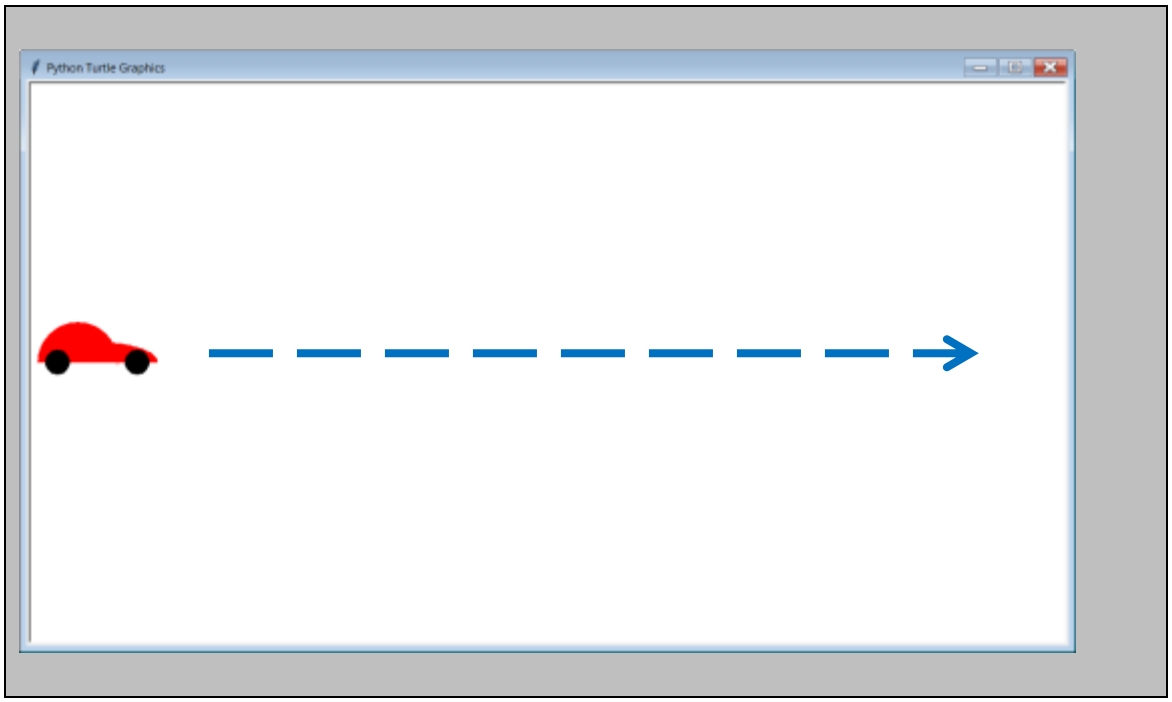
Finally, the front wheel moves across the screen.



Program **Animation10.py**, in Figure 18.35, properly animates the car across the screen. This time all 4 parts of the car are placed in the same **for** loop. For this to work, a little algebra is necessary as the **x** values of the *body*, *hood* and *front wheel* of the car are all relative to the **x** value of the *rear wheel*. While this program does fix the issue of the previous program, it also introduces a new one. When you execute the program, you see the entire car moving across the screen as we wish, but after a little while the car begins to slow down. It goes slower and slower and eventually drags to a crawl.

Figure 18.35

```
 1 # Animation10.py
 2 # This program properly animates the car across the
 3 # screen by placing all 4 shapes in the same <for> loop.
 4 # A little algebra is required since the location of 3
 5 # of the shapes is relative to the location of the rear
 6 # wheel.  This program also introduces another issue.
 7 # As more and more images are drawn/erased on/from the
 8 # screen, the output gets slower and slower and eventually
 9 # drags to a crawl.  And on top of all of this, on a Mac
10 # the car may leave behind a trail.
11
12
13
14 from Graphics import *
15
16
17 beginGrfx(1300,700)
18
19 for x in range(35,1176,5):
20    setColor("red")
21    fillArc(x+50,350,75,25,270,90) # draw body
22    fillArc(x+25,350,50,50,270,90) # draw hood
23    setColor("black")
24    fillCircle(x,350,15)           # draw rear wheel
25    fillCircle(x+100,350,15)       # draw front wheel
26    delay(10)
27    setColor("white")
28    fillArc(x+50,350,75,25,270,90) # erase body
29    fillArc(x+25,350,50,50,270,90) # erase hood
30    fillCircle(x,350,15)           # erase rear wheel
31    fillCircle(x+100,350,15)       # erase front wheel
32
33 endGrfx()
34
```



So, the animation works, but why does it get slower and slower? OK, this is going to get technical. When you update the graphics window (call the **update** command) it makes sure that everything is displayed. It does this by re-drawing everything since the last time the screen was cleared. When you look at program **Animation10.py**, you might notice that there is no call to the **update** procedure. Actually there is. You just do not see it because the **delay** procedure calls the **update** procedure. Keep in mind that while we are erasing the car many times, we are never clearing the screen. So this is what happens. The first time through the loop, the car is drawn on the left side of the screen. The second time through the loop, it actually draws the car twice. First at the original location again, and then, after erasing, it re-draws it 5 pixels over. The third time through the loop, the car is drawn 3 times. The fourth time through the loop, the car is drawn 4 times. This pattern maintains itself. Near the end of the loop, the car is being drawn more than 100 times. This is why it slows down so much.

The solution to this problem is actually something that will make the program even simpler. In the previous program, erasing the car is accomplished by re-drawing each individual piece of the car again in white. Program **Animation11.py**, in Figure 18.36, simply uses the **clear** command to erase the entire screen. Not only does this simplify the code, in that we are using a single command to erase the car instead of 4 separate commands, it also cures the slowing down problem. Remember I said that when you update the screen, it re-draws everything since the last time the screen was "cleared". Now that we are clearing the screen every time through the loop, the slowing down problem has been eliminated. Make sure you execute the program to see the difference.

Figure 18.36

```
 1 # Animation11.py
 2 # This program fixes all of the issues of the
 3 # previous program by using the <clear> command
 4 # to completely erase everything on the screen.
 5
 6
 7 from Graphics import *
 8
 9
10 beginGrfx(1300,700)
11
12 for x in range(35,1176,5):
13    setColor("red")
14    fillArc(x+50,350,75,25,270,90) # draw body
15    fillArc(x+25,350,50,50,270,90) # draw hood
16    setColor("black")
17    fillCircle(x,350,15)           # draw rear wheel
18    fillCircle(x+100,350,15)       # draw front wheel
19    delay(10)
20    clear()                        # erase everything
21
22 endGrfx()
23
```

Suppose you were never great at algebra. Maybe you actually believed those people who said that algebra is something you would "never use in real life." For those of you who are "mathematically challenged" I have an alternate way to animate complicated images like a car across the screen. This approach does not use any variables, so no algebra is required. The secret is to first draw the entire image. Then you delay and erase it. Then you completely re-draw the entire image a little over. Then you delay and erase again. Then draw everything for the third time a little more over, and then delay and erase. This should sound familiar. It is the same *draw-and-erase animation* that we have been using. The only difference is your program will be much longer because you are not taking advantage of the efficiency of the **for** loop. This is demonstrated by program **Animation12.py**, shown in Figure 18.37. Aside from being much longer, there is another issue with this program. The output is not as "smooth" as the previous program. That is because this program jumps the car forward **100** pixels at a

time, while the previous program moved only **5** pixels at a time. You may wonder why someone would write the program this way. Why do considerably more work to get something that does not look as good? The reason is that the shorter, better way is also more complicated. Those who understand it can take advantage of its efficiency. Those who do not, have no choice but to do things the long way.

Figure 18.37

```
 1 # Animation12.py
 2 # This program shows an alternate way to
 3 # animate the car across the screen.
 4 # While this way does not require Algebra
 5 # and is less complicated, the code is much
 6 # longer, and the output is not as smooth.
 7
 8
 9 from Graphics import *
10
11
12 beginGrfx(1300,700)
13
14 setColor("red")
15 fillArc(85,350,75,25,270,90)
16 fillArc(60,350,50,50,270,90)
17 setColor("black")
18 fillCircle(35,350,15)
19 fillCircle(135,350,15)
20 delay(1000)
21 clear()
22
23 setColor("red")
24 fillArc(185,350,75,25,270,90)
25 fillArc(160,350,50,50,270,90)
26 setColor("black")
27 fillCircle(135,350,15)
28 fillCircle(235,350,15)
29 delay(1000)
30 clear()
31
32 setColor("red")
33 fillArc(285,350,75,25,270,90)
34 fillArc(260,350,50,50,270,90)
35 setColor("black")
```

```
fillCircle(235,350,15)
fillCircle(335,350,15)
delay(1000)
clear()

setColor("red")
fillArc(385,350,75,25,270,90)
fillArc(360,350,50,50,270,90)
setColor("black")
fillCircle(335,350,15)
fillCircle(435,350,15)
delay(1000)
clear()

setColor("red")
fillArc(485,350,75,25,270,90)
fillArc(460,350,50,50,270,90)
setColor("black")
fillCircle(435,350,15)
fillCircle(535,350,15)
delay(1000)
clear()

setColor("red")
fillArc(585,350,75,25,270,90)
fillArc(560,350,50,50,270,90)
setColor("black")
fillCircle(535,350,15)
fillCircle(635,350,15)
delay(1000)
clear()

setColor("red")
fillArc(685,350,75,25,270,90)
fillArc(660,350,50,50,270,90)
setColor("black")
fillCircle(635,350,15)
fillCircle(735,350,15)
delay(1000)
clear()

setColor("red")
fillArc(785,350,75,25,270,90)
fillArc(760,350,50,50,270,90)
```

```
 80 setColor("black")
 81 fillCircle(735,350,15)
 82 fillCircle(835,350,15)
 83 delay(1000)
 84 clear()
 85
 86 setColor("red")
 87 fillArc(885,350,75,25,270,90)
 88 fillArc(860,350,50,50,270,90)
 89 setColor("black")
 90 fillCircle(835,350,15)
 91 fillCircle(935,350,15)
 92 delay(1000)
 93 clear()
 94
 95 setColor("red")
 96 fillArc(985,350,75,25,270,90)
 97 fillArc(960,350,50,50,270,90)
 98 setColor("black")
 99 fillCircle(935,350,15)
100 fillCircle(1035,350,15)
101 delay(1000)
102 clear()
103
104 setColor("red")
105 fillArc(1085,350,75,25,270,90)
106 fillArc(1060,350,50,50,270,90)
107 setColor("black")
108 fillCircle(1035,350,15)
109 fillCircle(1135,350,15)
110 delay(1000)
111 clear()
112
113 setColor("red")
114 fillArc(1185,350,75,25,270,90)
115 fillArc(1160,350,50,50,270,90)
116 setColor("black")
117 fillCircle(1135,350,15)
118 fillCircle(1235,350,15)
119 delay(1000)
120 clear()
121
122 endGrfx()
```

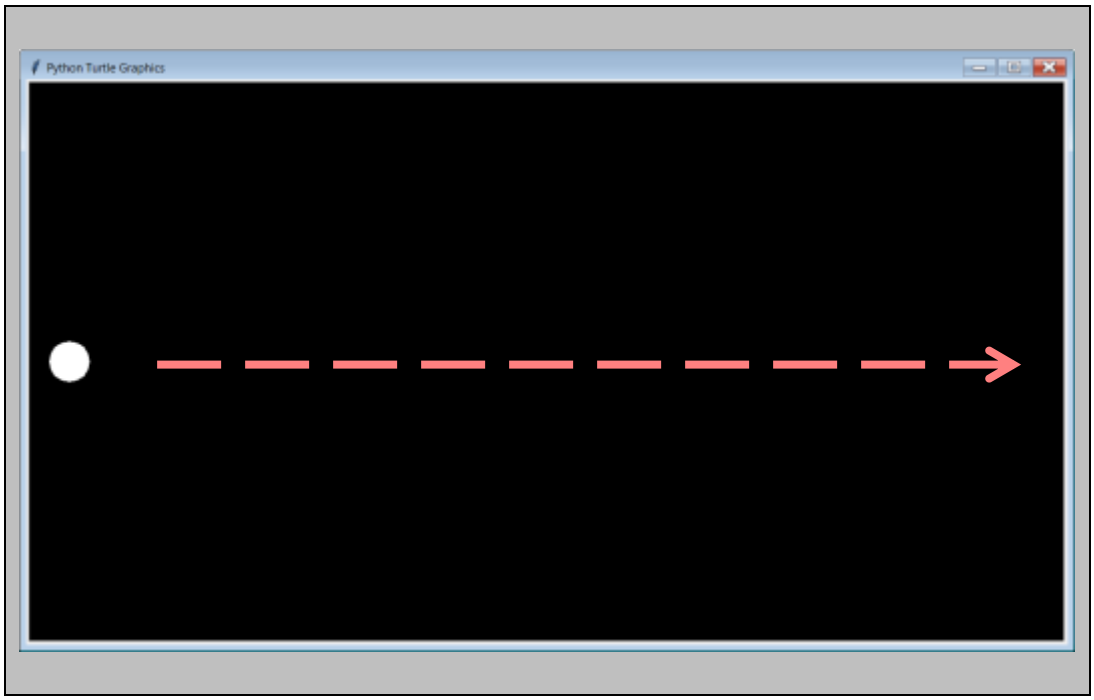
What if the background is not white?

In all of the previous animation programs, we have been using the color **"white"** to *erase* the images. The reason for this is simple. The color of the background was **"white"**. When doing *draw-and-erase animation*, the "erasing color" is always the always the same as the "background color".

Program **Animation13.py**, in Figure 18.38, is similar to program **Animation07.py**. The earlier program animated a black ball across a white screen. This program inverts the colors and now animated a white ball on a black screen. The key here is realizing that since the background is **"black"**, the "erasing color" is also **"black"**.

Figure 18.38

```
# Animation13.py
# This program is similar to Animation07.py,
# but now the circle is white and the
# background color is black.  This means
# <"black"> is now the "erasing color".


from Graphics import *


beginGrfx(1300,700)

setBackground("black")

for x in range(50,1251,1):
    setColor("white")
    fillCircle(x,350,25)
    delay(10)
    setColor("black")
    fillCircle(x,350,27)

endGrfx()

```

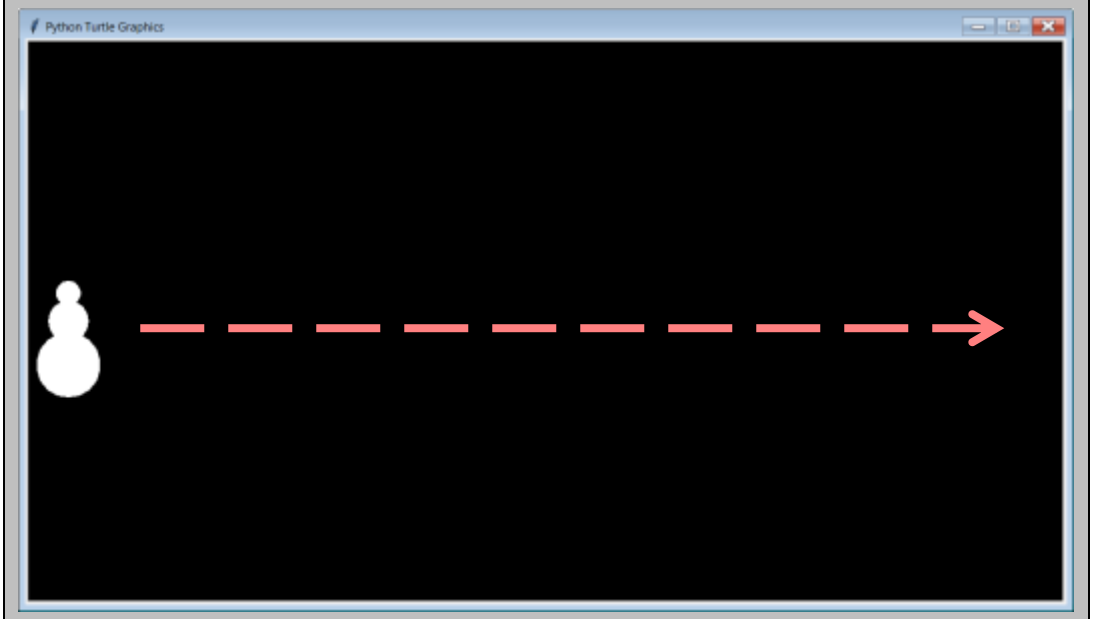


Instead of animating 1 circle across the screen, how about 3? Program **Animation14.py**, in Figure 18.39, draws and erases 3 circles inside the same **for** loop. While the circles are different sizes, and have different **y** values, there is no "algebra" required because the **x** values of the centers of all 3 circles are the same. You will notice that moving the snowman across the screen has the same "slowing down" problem that the car had earlier.

Figure 18.39

```
# Animation14.py
# This program animates a snowman across the screen
# by placing 3 circles in the same <for> loop.
# Note how the "slowing down" problem returns.


from Graphics import *


beginGrfx(1300,700)

setBackground("black")

```

```
14 for x in range(50,1251,1):
15     setColor("white")
16     fillCircle(x,315,15)
17     fillCircle(x,350,25)
18     fillCircle(x,405,40)
19     delay(10)
20     setColor("black")
21     fillCircle(x,315,17)
22     fillCircle(x,350,27)
23     fillCircle(x,405,42)
24
25 endGrfx()
26
```



The slowing down problem returned because of the way I am erasing the snowman. Each circle is being erased individually rather than simply using the **clear** command. The reason I did not use **clear** is that when the screen is cleared, it becomes white and I need it black. The solution is to go ahead a **clear** the screen, but then immediately set the background color to "**black**". This is done in program **Animation15.py**, in Figure 18.40. This is an example of *draw-and-redraw animation*.

Figure 18.40

```
 1 # Animation15.py
 2 # This program fixes the "slowing down" problem
 3 # of the previous program by using <clear>.
 4 # This does require that the background color
 5 # is set immediately after the <clear> command.
 6 # NOTE: This is called "Draw-And-Redraw Animation".
 7
 8
 9 from Graphics import *
10
11
12 beginGrfx(1300,700)
13
14 for x in range(50,51,3):
15    clear()
16    setBackground("black")
17    setColor("white")
18    fillCircle(x,315,15)
19    fillCircle(x,350,25)
20    fillCircle(x,405,40)
21    delay(10)
22
23 endGrfx()
24
```

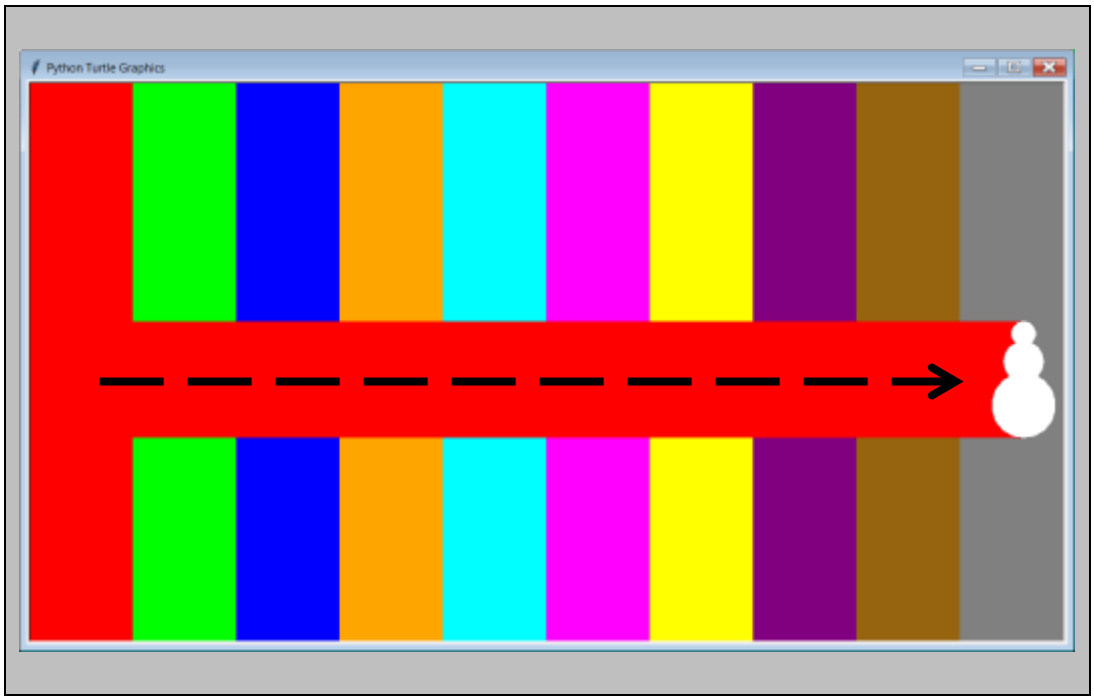
Animation with a Multi-Colored Background

With *draw-and-erase* animation you follow these simple steps: First, you draw something. Then, you wait a tiny bit of time. Then, you erase it by drawing over it in the background color. Then you re-draw it in a different location and repeat the process. OK, but what if your background is not one solid color? What if you have a multi-colored background? This problem is demonstrated with program **Animation16.py** in Figure 18.41. The program starts with a call to procedure **drawColorfulBackground** which generates 9 different colored vertical stripes.

After that, a **for** loop is used to *move* the snowman across the screen using the same *draw-and-erase animation* from program **Animation14.py**. Since the first stripe is **red**, **red** is used as the "erasing color". When you execute the program it seems to work at first, but once the snowman gets beyond the first stripe, the animation ceases to work properly. Keep in mind that this is an example of what NOT to do. On top of everything else, since we are using *draw-and-redraw animation* instead of *draw-and-redraw animation*, we have brought back the "slowing down" issue.

Figure 18.41

```
# Animation16.py
# This program shows another major problem with
# "Draw-And-Erase Animation."  It falls flat on
# its face when you have a multi-colored background.


from Graphics import *


def drawColorfulBackground():
    for c in range(10):
        setColor(c+1)
        fillRectangle(c*130, 0, (c+1)*130, 700)



##########
#  MAIN  #
##########

beginGrfx(1300,700)

drawColorfulBackground()

for x in range(50,1251,2):
    setColor("white")
    fillCircle(x,315,15)
    fillCircle(x,350,25)
    fillCircle(x,405,40)
    delay(10)
    setColor("red")
    fillCircle(x,315,17)
    fillCircle(x,350,27)
    fillCircle(x,405,42)

endGrfx()
```



Aside from the "slowing down" issue, *draw-and-erase animation* falls completely on its face when you have a multi-colored background. While it may seem counter-intuitive to redraw the entire background every time something moves; in Python at least, *draw-and-redraw animation* works much better.

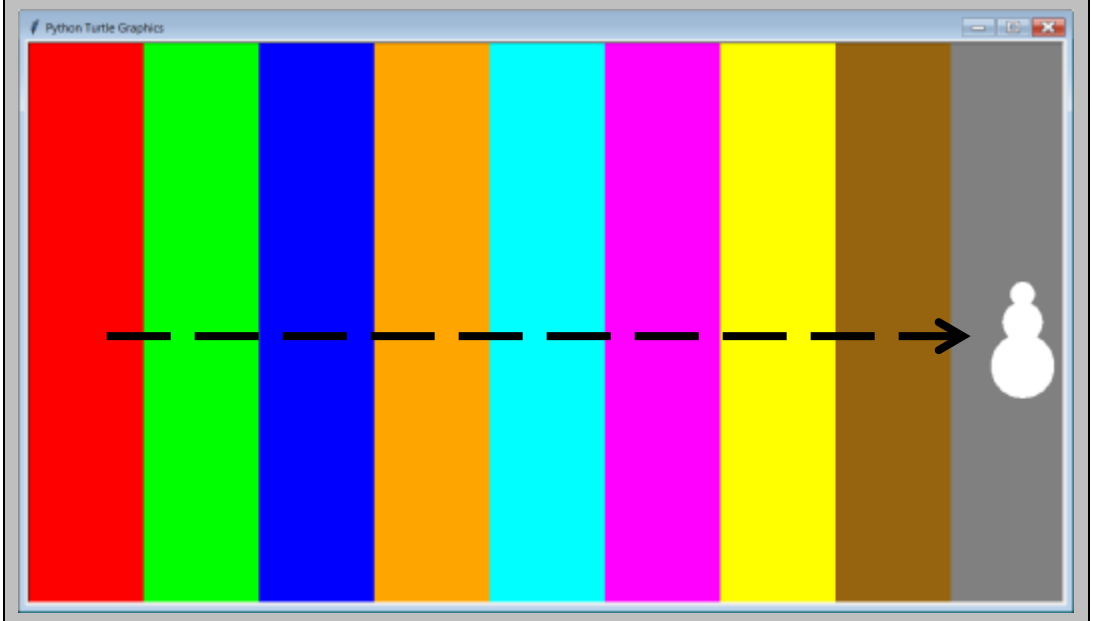
Program **Animation17.py**, in Figure 18.42, shows how *draw-and-redraw animation* properly moves the snowman across the multi-colored background. You should notice two big differences between this program and **Animation16.py**. First, the procedure call to **drawColorfulBackground** is now inside the **for** loop that moves the snowman. Second, the snowman is never erased. Every time the snowman moves the entire background is re-drawn on top of him.

NOTE: When using *draw-and-redraw animation* with a multi-colored background, it is strongly recommended that you first create a separate procedure to draw your background.

Figure 18.42

```
1 # Animation17.py
2 # This program shows how to handle a multi-colored
3 # background. By using "Draw-And-Redraw Animation"
4 # instead of "Draw-And-Erase" the entire background
5 # is redrawn every time the object moves.
6
7
```

```
 8 from Graphics import *
 9
10
11 def drawColorfulBackground():
12    for c in range(10):
13       setColor(c+1)
14       fillRectangle(c*130, 0, (c+1)*130, 700)
15
16
17
18 ##########
19 #  MAIN  #
20 ##########
21
22 beginGrfx(1300,700)
23
24 for x in range(50,1251,2):
25    clear()
26    drawColorfulBackground()
27    setColor("white")
28    fillCircle(x,315,15)
29    fillCircle(x,350,25)
30    fillCircle(x,405,40)
31    delay(10)
32
33 endGrfx()
```



Draw and Redraw Animation: Step-By-Step Review

1. Clear the screen.
2. Draw the entire background (preferably by using a procedure).
3. Draw the foreground image.
4. Delay for a fraction of a second.
5. Repeat this process over and over, each time drawing the foreground image in a slightly different location.

This is the recommended approach for Computer Animation in our class because it solves BOTH of the problems of *Draw and Erase Animation*:

1. It works even if your background is not one solid color.
2. The animation will not slow down, even for complicated images.

18.7 The Last Word

The final section in this graphics chapter actually has nothing to do with graphics at all. This is the final section of the final chapter of this textbook, and there is one last point I want to make – and now you are ready to understand it.

Some people may be concerned that with technology changing so rapidly, the programming knowledge that they have acquired in this course will soon be out of date. While new programming languages pop up from time to time, the logic of programming has never changed.

The next 7 programs all do the exact same thing. The first is written in Python. The second is in Java. The third is in C++; the fourth is in Pascal; the fifth is in BASIC; the sixth is written in Graphical ROBOT-C and the seventh is written in VEXcode VR Blocks. Even though you may have never seen some of these languages before, you may be surprised how familiar these programs will look to you now that you know Python.

Program **LastWord01.py**, in Figure 18.43 shows nothing new. There is a **for** loop that counts backwards from **20** down to **0**. Inside the **for** loop is an **if…else** structure that checks if the loop counter is greater than or equal to **10**. If it is, it displays that the loop counter "**is a 2-digit number.**" Otherwise, it displays that the loop counter "**is a 1-digit number.**" The intent here is not to teach any new material. You learned about *control structures* some time ago. The intent is to have you compare this program to the other five.

Figure 18.43

```
 1 # LastWord01.py
 2 # This program will count backwards from 20 to 0.
 3 # For each number, it will display whether it has
 4 # 1 or 2 digits. Compare this Python code to the
 5 # code from other languages.
 6
 7
 8 print()
 9 for k in range (20, -1, -1):
10     if k >= 10:
11         print (k,"is a 2-digit number.")
12     else:
13         print (k,"is a 1-digit number.")
```

```
----jGRASP exec: python LastWord01.py

20 is a 2-digit number.
19 is a 2-digit number.
18 is a 2-digit number.
17 is a 2-digit number.
16 is a 2-digit number.
15 is a 2-digit number.
14 is a 2-digit number.
13 is a 2-digit number.
12 is a 2-digit number.
11 is a 2-digit number.
10 is a 2-digit number.
9 is a 1-digit number.
8 is a 1-digit number.
7 is a 1-digit number.
6 is a 1-digit number.
5 is a 1-digit number.
4 is a 1-digit number.
3 is a 1-digit number.
2 is a 1-digit number.
1 is a 1-digit number.
0 is a 1-digit number.

----jGRASP: operation complete.
```

Now look at program **LastWord02.java**, in Figure 18.44. This is the same program written in Java. Keep in mind that Java programs can also be loaded and executed in **jGRASP**. Even if you may have never seen a Java program before, you may be surprised just how familiar the program will look, even though it is written in a different language. Note how both programs have commands to display text. Both have selection and repetition control structures. Both have relational operators. Both even have comments. Python may use a hashtag (#) and Java may use a double-slash (//), but both languages still have comments.

Figure 18.44

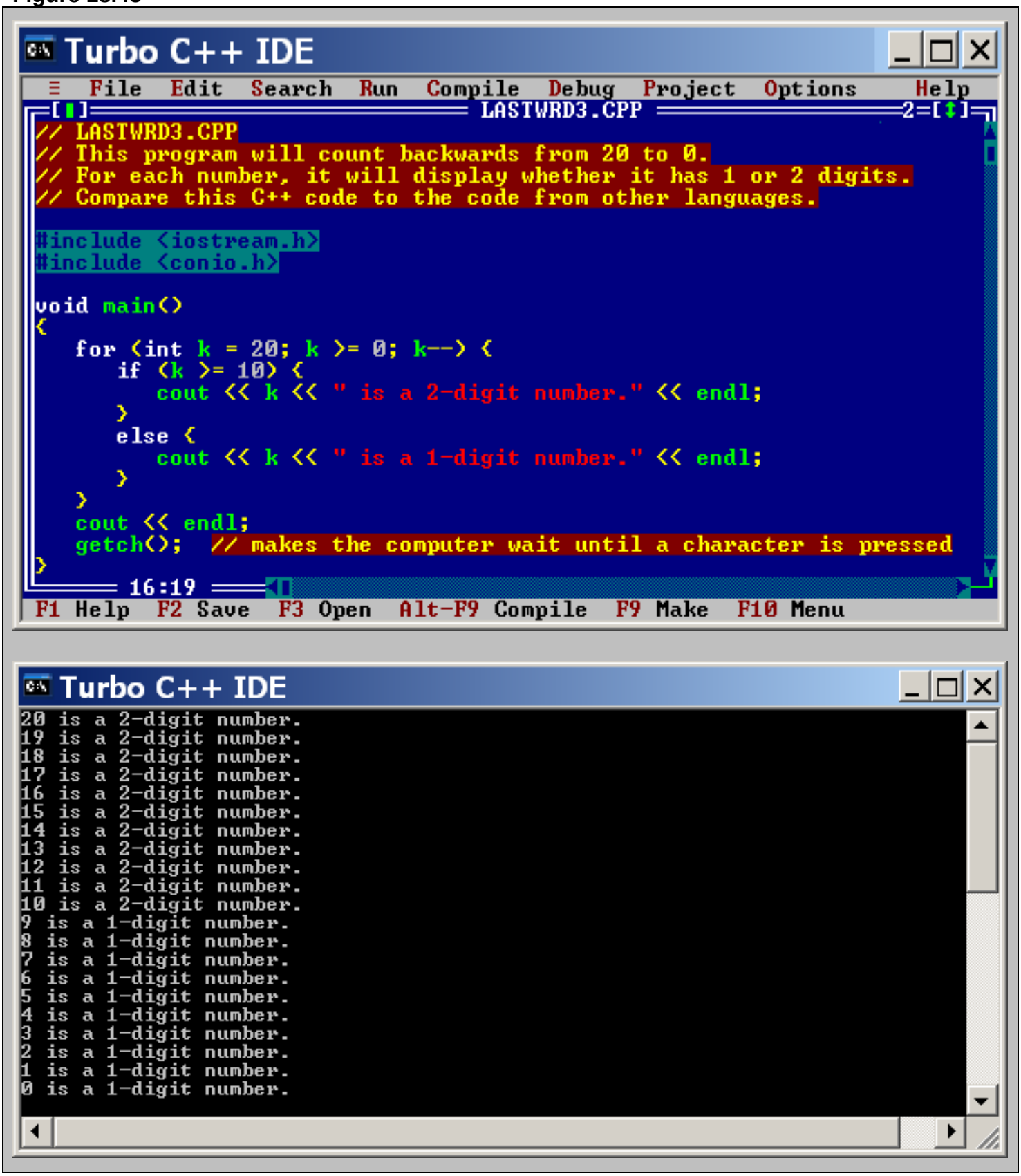
```
 1 // LastWord02.java
 2 // This program will count backwards from 20 to 0.
 3 // For each number, it will display whether it has
 4 // 1 or 2 digits. Compare this Java code to the
 5 // code from other languages.
 6
 7 public class LastWord02
 8 {
 9    public static void main (String args[])
10    {
11       System.out.println();
12       for (int k = 20; k >= 0; k--)
13       {
14          if (k >= 10)
15          {
16             System.out.println(k + " is a 2-digit number.");
17          }
18          else
19          {
20             System.out.println(k + " is a 1-digit number.");
21          }
22       }
23    }
24 }
25
```

The output is identical to the Python program.

Now look at program **LASTWRD3.CPP**, in Figure 18.45. You can load this program, and the next couple, in **jGRASP (**because they are text files), but you will not be able to execute them. You may wonder why it seems that a couple characters are missing from the filename. This is because this C++ file is from the old DOS days. Back then, file names had an "8.3 format". This means the *prefix* part of the filename (the part before the "dot") could not have more than 8 characters and the *extension* (the part after the "dot") could not have more than 3.

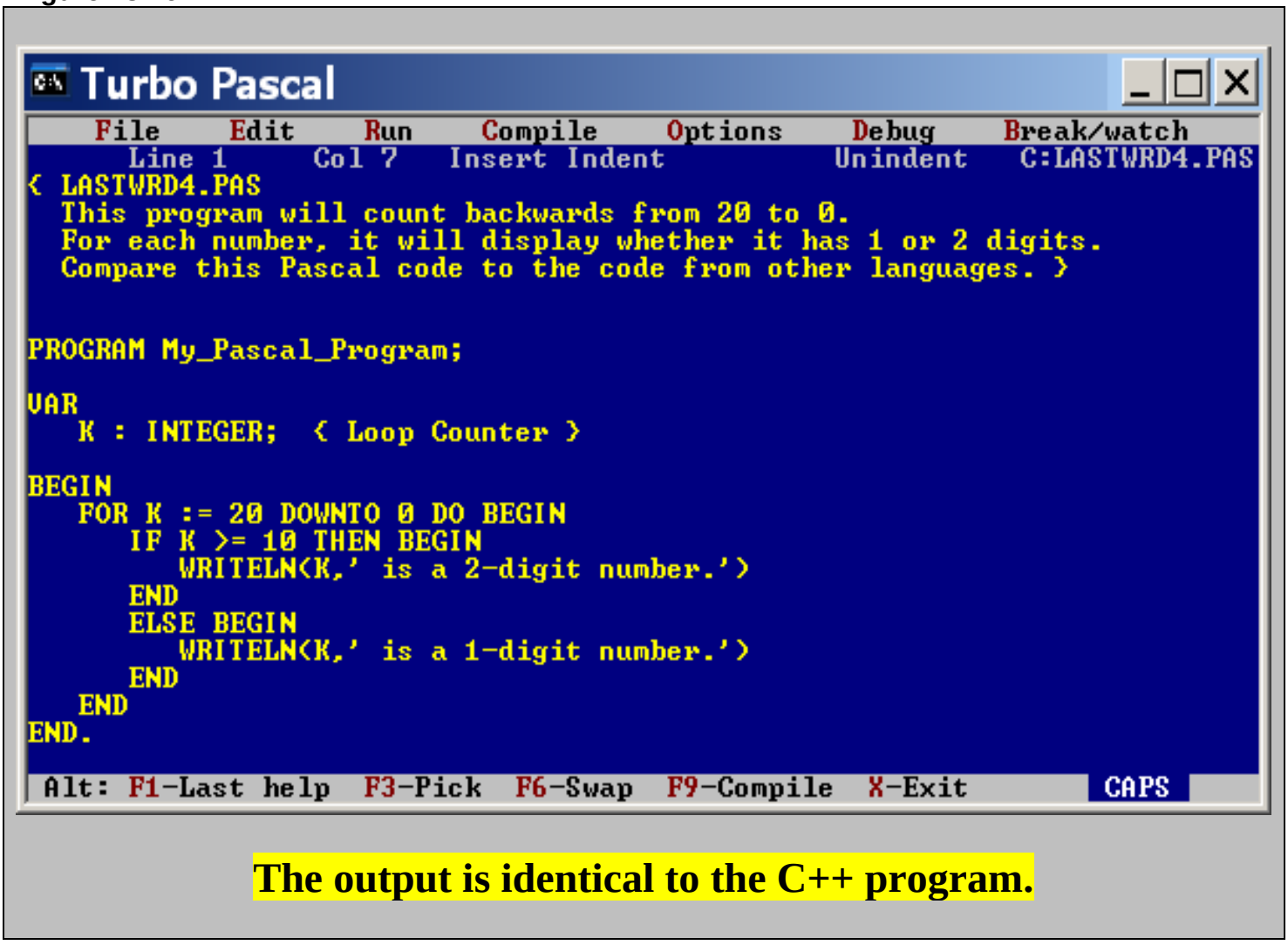
OK, let us focus on the C++ program itself. As with the Java program, even though you may have never seen a C++ program before, it should still look familiar. In fact, since you just saw the same program written in Java, it should look very familiar. The creators of Java copied quite a bit from C++. The **for** loop, the **if..else** structure, the use of braces and even the *comments* are identical.

Figure 18.45


Turbo C++ IDE
File Edit Search Run Compile Debug Project Options Help
LASTWRD3.CPP
// LASTWRD3.CPP
// This program will count backwards from 20 to 0.
// For each number, it will display whether it has 1 or 2 digits.
// Compare this C++ code to the code from other languages.
#include <iostream.h>
#include <conio.h>
void main()
{
for (int k = 20; k >= 0; k--) {
if (k >= 10) {
cout << k << " is a 2-digit number." << endl;
}
else {
cout << k << " is a 1-digit number." << endl;
}
}
cout << endl;
getch(); // makes the computer wait until a character is pressed
}
16:19
F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu
Turbo C++ IDE
20 is a 2-digit number.
19 is a 2-digit number.
18 is a 2-digit number.
17 is a 2-digit number.
16 is a 2-digit number.
15 is a 2-digit number.
14 is a 2-digit number.
13 is a 2-digit number.
12 is a 2-digit number.
11 is a 2-digit number.
10 is a 2-digit number.
9 is a 1-digit number.
8 is a 1-digit number.
7 is a 1-digit number.
6 is a 1-digit number.
5 is a 1-digit number.
4 is a 1-digit number.
3 is a 1-digit number.
2 is a 1-digit number.
1 is a 1-digit number.
0 is a 1-digit number.

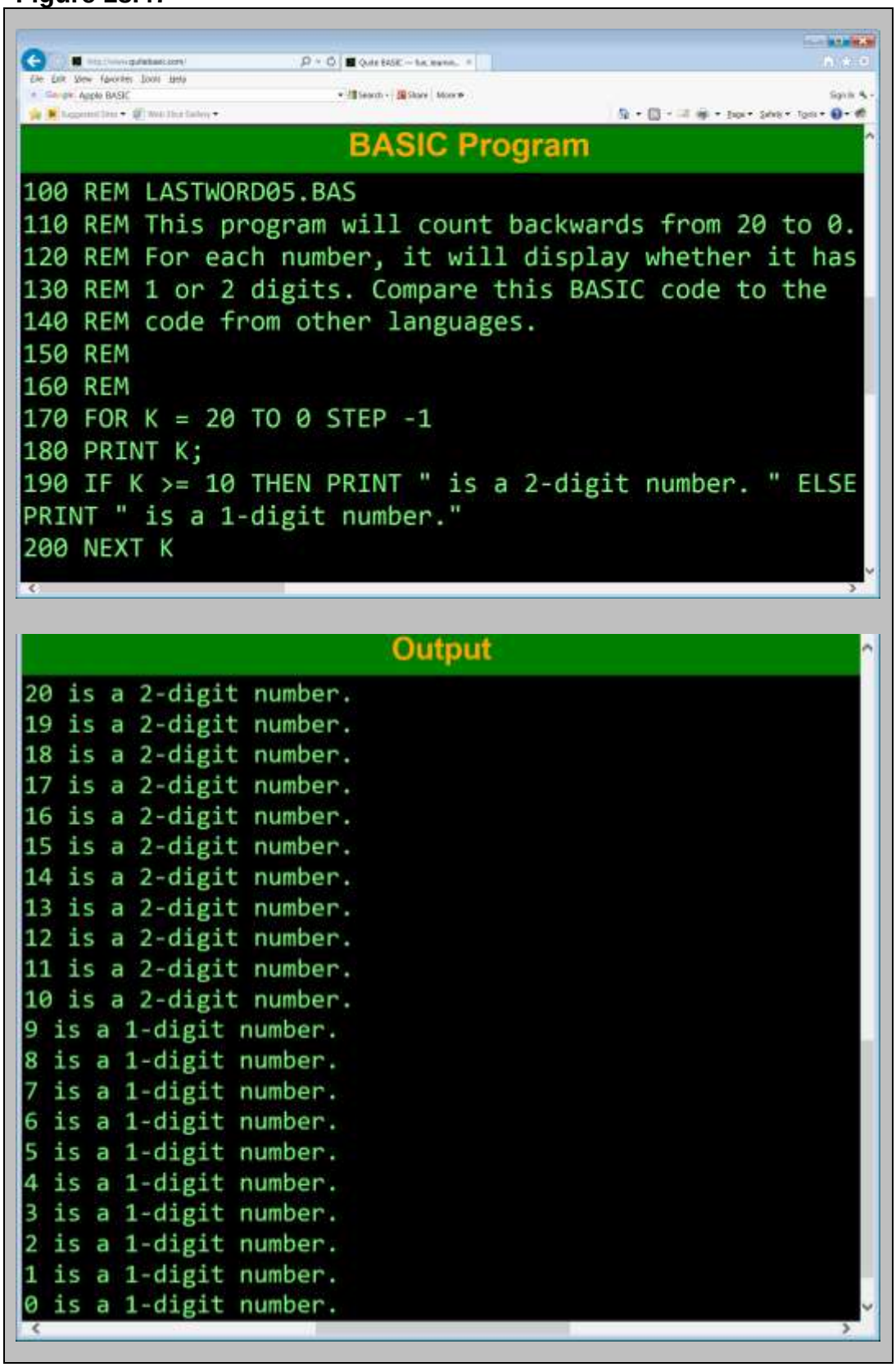
Borland, the company that made **Turbo C++**, also made **Turbo Pascal** several years earlier. Let us look at program **LASTWRD4.PAS**, shown in Figure 18.46. This is the same program written in Pascal. Compared to Java and C++, Pascal is an older language, so there are some definite differences. You will note that **Turbo Pascal** does not use different colors for different parts of the program. That feature came later. Braces { } mark the beginning and ending of blocks in Java and C++. In Pascal the words **BEGIN** and **END** are used instead. Braces are still used in Pascal, but they used for comments.

Figure 18.46



For our penultimate (second to last) program, let us go back even further, to the first language that I ever learned, and look at program **LASTWORD05.BAS**, in Figure 18.47. For the C++ and Pascal examples, I was able to execute the programs from an old Windows XP laptop. For BASIC, I had to use a web-based emulator. The one I use is **http://www.quitebasic.com**.

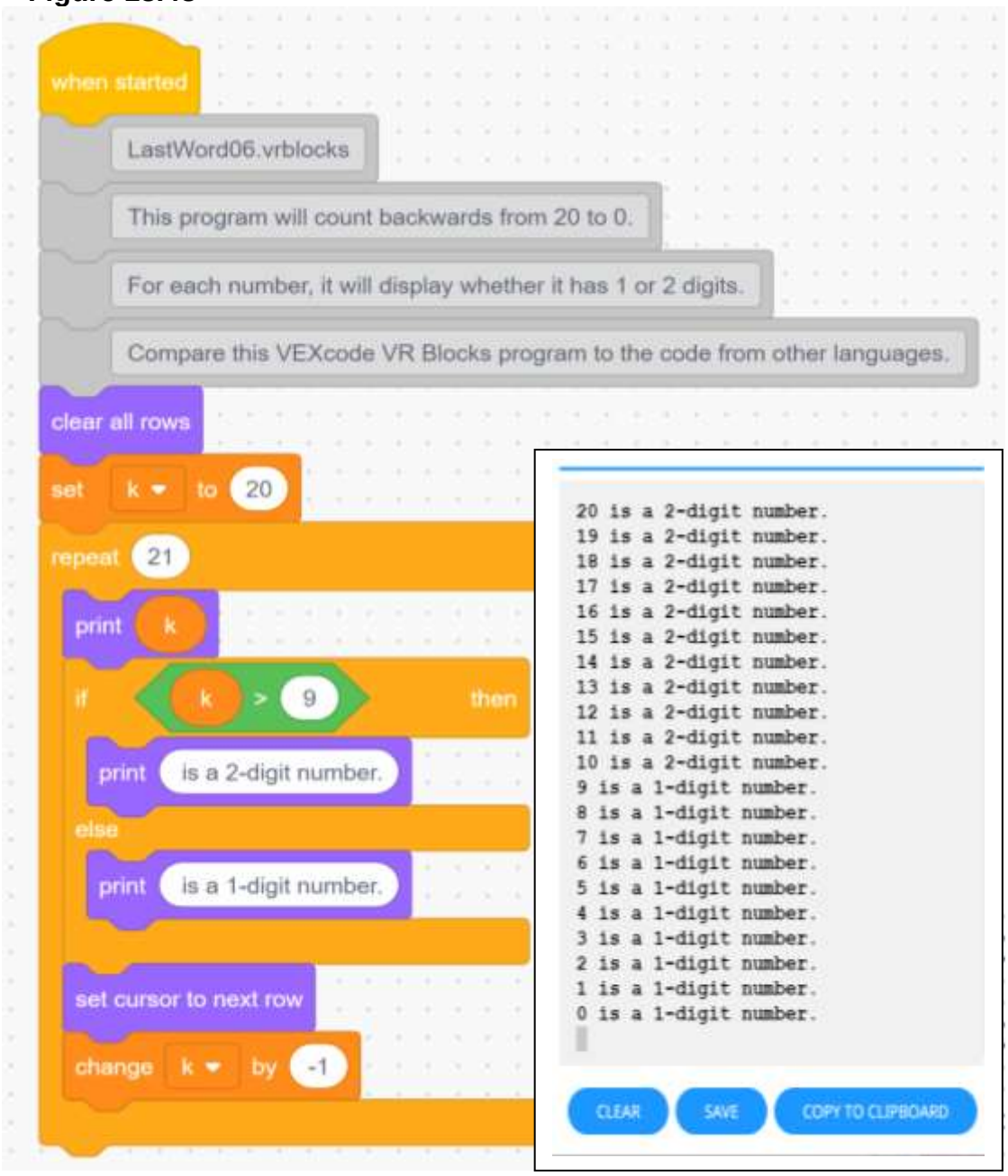
Figure 18.47


BASIC Program
100 REM LASTWORD05.BAS
110 REM This program will count backwards from 20 to 0.
120 REM For each number, it will display whether it has
130 REM 1 or 2 digits. Compare this BASIC code to the
140 REM code from other languages.
150 REM
160 REM
170 FOR K = 20 TO 0 STEP -1
180 PRINT K;
190 IF K >= 10 THEN PRINT " is a 2-digit number. " ELSE
PRINT " is a 1-digit number."
200 NEXT K
Output
20 is a 2-digit number.
19 is a 2-digit number.
18 is a 2-digit number.
17 is a 2-digit number.
16 is a 2-digit number.
15 is a 2-digit number.
14 is a 2-digit number.
13 is a 2-digit number.
12 is a 2-digit number.
11 is a 2-digit number.
10 is a 2-digit number.
9 is a 1-digit number.
8 is a 1-digit number.
7 is a 1-digit number.
6 is a 1-digit number.
5 is a 1-digit number.
4 is a 1-digit number.
3 is a 1-digit number.
2 is a 1-digit number.
1 is a 1-digit number.
0 is a 1-digit number.

BASIC is even older than Pascal. This goes back to the days before structured programming. Note that the entire **IF…THEN…ELSE** structure is typed in one word-wrapping line. Even so, we see the same components exist here that existed in all of the other program language examples. If you are wondering why the comments are preceded with the command **REM**, it is because comments used to be called "remarks".

Our final program example for this section, chapter and for the entire textbook is **LastWord06.vrblocks**, shown in Figure 18.48.

Figure 18.48



In 2020, VEX Robotics released *VEXcode VR Blocks* which allows you to program a *virtual robot* online. VEXcode VR Blocks is based on the language *Scratch*. Note how the blocks fit together. The basic philosophy is that if a block does not fit there, then it does not belong there. Note how number variables have an oval shape while the Boolean expression is hexagonal. This makes is impossible to put a Boolean variable or expression where a number variable or expression belongs. It makes the reverse true as well.

The point being made with these 6 programs is that while the syntax is different with different languages, the logic is the same. All of these languages can display text. Python, BASIC and VEXcode VR Blocks use **print**. Java uses **System.out.println**. C++ uses **cout**. Pascal uses **WRITELN**. All of these languages also have comments and variables and repetition structures and selection structures and relational operators.

I will use one final example to emphasize this point. At John Paul II High School, we had a religion teacher who is known as "Sister Peggy". In 2007, Sister Peggy came by my classroom to drop something off and noticed the assignment my students were doing. "Oh, they are learning the **for** loop" she said. I was a little surprised, until I learned that Sister Peggy taught Computer Science way back in the 1970s. Back then, she taught the language *FORTRAN*. While FORTRAN is very different from Python, the logic is still the same. I will conclude this chapter and this textbook with this statement:

Programming knowledge will never become obsolete!