Chapter VIII

Repetition Control Structures and Random Numbers

Chapter VIII Topics

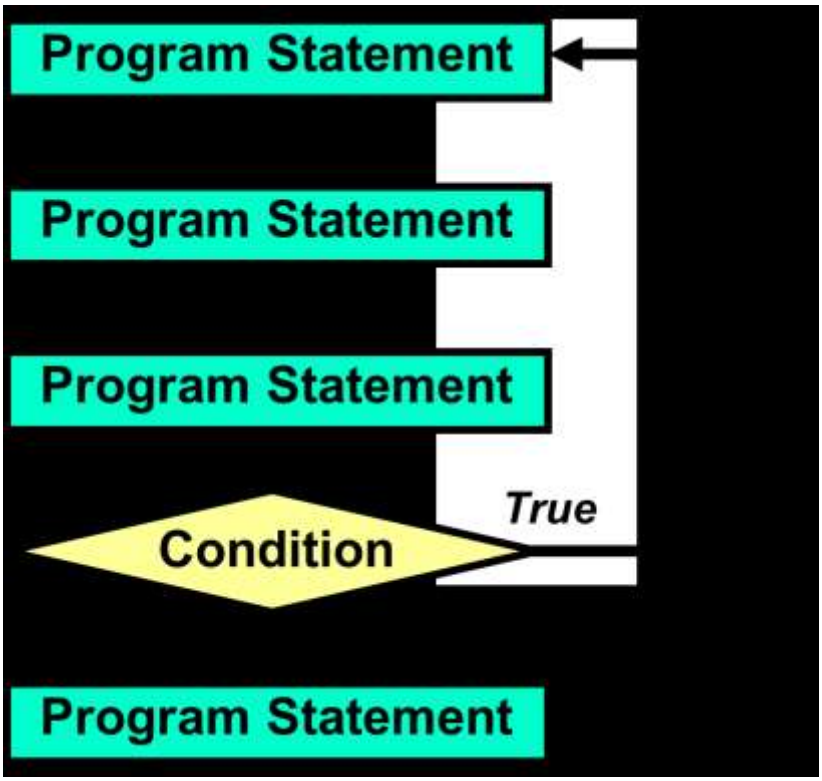
8.1 Introduction

Chapter VII introduces the concepts of *Program Flow* and *Control Structures* and how the latter can affect the former. You may remember this definition:

Program Flow Review

Program Flow follows the exact sequence of listed program statements, unless directed otherwise by a Python control structure.

While most of Chapter VII was devoted to *Selection Control Structures* you were at least shown a diagram that represents *Repetition Control Structures*.



In the diagram, you see that *Repetition Control Structures*, just like *Selection Control Structures* have a *condition* and a certain path that is only followed if that condition is **True**.

8.2 Fixed Repetition

When talking about *repetition*, some people like the simpler term *looping*. Other prefer the more formal term *iteration*. The name does not matter, as long as you realize that this is a control structure that executes certain program segments repeatedly.

The Python language has two different types of *repetition* control structures. In this section we will explore *fixed repetition* with the **for** loop structure. *Fixed* means that you know – ahead of time – how many times a certain program segment will repeat. You can determine this just by looking at the code of the loop structure. It is not necessary to run the program to find this out.

Before we actually look at a Python loop structure, let us first look at a program that helps to motivate the need for efficiency. The next 2 programs display 20 advertising lines for Joe's Friendly Diner. Now I do not care, and hopefully you do not care why Joe wants to display the following message 20 times:

```
Eat at Joe's friendly diner for the best lunch value!
```

The point is that we need to write a program that will generate that message. Joe is paying us good money, and we are just lowly programmers following the orders of a customer.

There is one way to accomplish this task with the knowledge you have acquired so far. You can write 20 different **print** statements. Program **Repetition01.py**, in Figure 8.1 demonstrates the inefficient repetition without using any loop structures. That approach is not really so bad, and with a little clever block copying, you are quickly done in no time. This is true, but Joe only wants 20 lines. What about a program that needs to print 100 or 1000 or 1,000,000 flyers for Joe's friendly diner? Are you then going to generate 1,000,000 program statements that each displays the same thing? You see the point here. There must be a better way. Please do not ever use this approach.

Figure 8.1

```
1 # Repetition01.py
2 # This program displays 20 identical lines of text.
3 # The program is very inefficient in that it uses
4 # 20 separate <print> statements.
5
6
7 print()
8 print("Eat at Joe's Friendly Diner for the best lunch value!")
9 print("Eat at Joe's Friendly Diner for the best lunch value!")
```

```python
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
print("Eat at Joe's Friendly Diner for the best lunch value!")
```

```
 ----jGRASP exec: python Repetition01.py

Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!
Eat at Joe's Friendly Diner for the best lunch value!

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

We apologize for that last horrendous program. It was necessary since we want to make a strong point why loop structures are very beneficial in computer science. The same *Joe's Diner* program is going to be demonstrated again. However, this time the program uses a **for** loop control structure to assist with the repetition process. Program **Repetition02.py**, in Figure 8.2, has the exact same output as the previous program example, but is considerably shorter.

Figure 8.2

```
 1 # Repetition02.py
 2 # This program displays 20 identical lines of text like
 3 # the last program, but is much more efficient because
 4 # is uses a <for> loop.
 5
 6
 7 print()
 8
 9 for k in range(20):
10   print("Eat at Joe's Friendly Diner for the best lunch value!")
11
```

How is the repetition accomplished in this program? Imagine you did not have a computer and you were told to write this message 20 times by hand. What would you have to do to make sure you wrote the message exactly 20 times, and not 19 or 21 times? You would need to count. A fixed loop is one that repeats a "fixed" number of times. For this to work, the loop needs a *loop counter*. The fancy term for this is a **L***oop* **C***ontrol* **V***ariable* or LCV. In this program, **k** is the *loop counter*. The **20** used with **range** indicates this loop will repeat **20** times.

NOTE: It is considered "OK" to use a single letter variable for a loop counter.
ALSO: This is one of the very few times that it is considered "OK" to use a single letter variable.

Before we go on, we need to review another important concept that was first introduced in Chapter VII. Program **Repetition03.py**, in Figure 8.3, is almost identical to the previous program. The only different is the **print** statement is not indented. As with selection control structures, when you do not use proper indenting with repetition control structures, the program cannot execute.

Figure 8.3

```
 1 # Repetition03.py
 2 # This program demonstrates the Syntax Error
 3 # you receive when you do not properly indent
 4 # the programming statement(s) being controlled
 5 # by a control structure.
 6
 7 # NOTE: In most languages, indentation is recommended.
 8 #       In Python, indentation is required.
 9
10
11 for k in range(20):
12 print("Eat at Joe's Friendly Diner for the best lunch value!")
13
```

```
 ----jGRASP exec: python Repetition03.py
  File "Repetition03.py", line 12
    print("Eat at Joe's Friendly Diner for the best lunch value!")
    ^
IndentationError: expected an indented block after 'for' statement
on line 11

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

Indentation Rule Review

In most languages, indenting the program statements that are “controlled” by control structures is recommended.

In Python, it is required.

Python programs that do not use proper and consistent indentation will not execute.

So, let us look again at the statement

```
for k in range(20):
```

We already mentioned that this loop repeats **20** times and that **k** is the *loop counter*. Does this mean that **k** counts from 1 to 20? That would seem logical, but let us test it to be sure. Program **Repetition04.py**, in Figure 8.4, uses the same **for** loop heading as before; however, this time the program is not displaying an advertisement for Joe's restaurant. Instead, it is displaying the value of the loop counter itself. Will we see the numbers from 1 to 20? Run the program to find out.

Figure 8.4

```
1 # Repetition04.py
2 # This program displays the value of the loop counter
3 # which is also called the "Loop Control Variable" (LCV).
4 # Note that even though the loop repeats 20 times,
5 # the counter actually counts from 0 to 19.
6
7
8 print()
9
10 for k in range(20):
11     print(k, end = " ")
12
13 print()
14
```

```
----jGRASP exec: python Repetition04.py

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

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

Surprisingly, we do not see the numbers 1 to 20. Instead, we see the numbers **0** to **19**. Keep in mind, the loop still repeats **20** times. You just have to remember that in Python, by default, **for** loop counters start their *counting* at **0**. Also, by default, **for** loop counters *count* by **1**s.

Program **Repetition05.py**, in Figure 8.5, demonstrates how to repeat multiple statements in a **for** loop. As you have already seen with *selection*, the *repetition* of multiple program statements works fine as long as proper, consistent indentation is used.

Figure 8.5

```
1 # Repetition05.py
2 # This program demonstrates that multiple program
3 # statements can be controlled with a <for>
4 # loop structure as long as proper,
5 # consistent indentation is used.
6
7
8 print()
9
10 for k in range(10):
11     print("########################")
12     print("##   Box Number",k,"  ##")
13     print("########################")
14     print()
15
```

```
----jGRASP exec: python Repetition05.py

######################
##   Box Number 0   ##
######################

######################
##   Box Number 1   ##
######################

######################
##   Box Number 2   ##
######################

######################
##   Box Number 3   ##
######################
```

```
######################
##   Box Number 4   ##
######################

######################
##   Box Number 5   ##
######################

######################
##   Box Number 6   ##
######################

######################
##   Box Number 7   ##
######################

######################
##   Box Number 8   ##
######################

######################
##   Box Number 9   ##
######################


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

Earlier, it was mentioned that "in Python, **for** loop counters start counting at **0** by default." That means, they do not have to start counting at **0**. We can change that if we want. All we need to do is use two numbers in the **range** command. The counter will begin with the first number and stop before it reaches the second number. Program **Repetition06.py**, in Figure 8.6, demonstrates a couple examples of this. In the output, you will see that the second number is never actually reached. In the second **for** loop, I intentionally added 1 to the second number to make the loop stop on **30** instead of **29**.

Figure 8.6

```
 1 # Repetition06.py
 2 # This program demonstrates how to make the <for>
 3 # loop start counting at a number other than zero.
 4 # The secret is to use 2 numbers in the <range> command.
 5 # The counter will begin with the first number.
 6 # and stop before it reaches the second number.
 7
 8
 9 print()
10
11 for k in range(10,30): # Displays 10 to 29
12     print(k, end = " ")
13
14 print("\n")
15
16 for k in range(10,31): # Displays 10 to 30
17     print(k, end = " ")
18
19 print()
20
```

```
----jGRASP exec: python Repetition06.py

10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

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

We also mentioned earlier that the **for** loop counts "by **1**s by default". Again, this means it does not have to. We can add a third number to the **range** command to change the *step* value. Program **Repetition07.py**, in Figure 8.7, demonstrates this with another two **for** loop structures.

The first two loops display even numbers only. This is because they both have a *step* value of **2**.

Figure 8.7

```
 1 # Repetition07.py
 2 # This program demonstrates how to change the "step"
 3 # value in the <for> loop.  By default, it is 1.
 4 # To count by a number other than 1 requires adding
 5 # a third number to the <range> command.
 6 # NOTE: As before, you may need to add 1 to the
 7 # "stopping value" to make the loop work properly.
 8
 9
10 print()
11
12 for k in range(10,30,2): # Displays evens from 10-28
13    print(k, end = " ")
14
15 print("\n")
16
17 for k in range(10,31,2): # Displays evens from 10-30
18    print(k, end = " ")
19
20 print()
21
```

```
  ----jGRASP exec: python Repetition07.py

 10 12 14 16 18 20 22 24 26 28

 10 12 14 16 18 20 22 24 26 28 30

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

Repetition08.py, in Figure 8.8, demonstrates that we can actually count by any amount. We can even count backwards. Four more **for** loops are shown.

The first loop counts from **5** to **100** by **5**s. (Like when you play hide and seek: "5, 10, 15, 20, 25, 30 … 95, 100. Ready or not, here I come!)

The second loop counts from **50** to **80** by 3s. This displays the numbers: **50**, **53**, **56**, **59**, **62**, **65**, **68**, **71**, **74**, **77** and **80**. Note that these are not actually multiples of **3**. This is because the starting number was not a multiple of **3**. Think about this, in the first loop, what would happen if the first number was odd instead of even? You would actually display only odd numbers in this case.

The last two loops demonstrate that if the *step* value is negative, the loop actually counts backwards. This is good for count downs. Notice that even when counting backwards, the computer stops before it reaches the second number. In the last loop, I want a count down from **20** all the way to **0**. For this reason, I needed to make the second number **-1**.

Figure 8.8

```
 1 # Repetition08.py
 2 # This program demonstrates that you can count by any
 3 # number and even count backwards.
 4
 5
 6 print()
 7
 8 for k in range(5,101,5): # Displays 5 to 100 by 5s
 9    print(k, end = " ")
10 print("\n")
11
12 for k in range(50,81,3): # Displays 50 to 80 by 3s
13    print(k, end = " ")
14 print("\n")
15
16 for k in range(20,0,-1): # Displays 20 down to 1
17    print(k, end = " ")
18 print("\n")
19
20 for k in range(20,-1,-1): # Displays 20 down to 0
21    print(k, end = " ")
22 print()
```

```
 ----jGRASP exec: python Repetition08.py

5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100

50 53 56 59 62 65 68 71 74 77 80

20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

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

Fixed Repetition

Fixed Repetition is done with the **`for`** loop structure.

General Syntax:

for LCV in range(Start,Stop,Step):
    execute program statement(s)

The *LCV* is the *Loop Control Variable* or *Loop Counter*.

Start specifies the first counting value.
If not specified, the default value is **0**.

The loop will "stop" before it reaches the *Stop* value.

Step specifies what the loop "counts by".
If not specified, the default value is **1**.

Other computer science terms for *repetition* are *looping* and *iteration*.

Specific Examples:

```
for k in range (10):          # Counts from 0 to 9
   print(k, end = " ")

for k in range (10,20):       # Counts from 10 to 19
   print(k, end = " ")

for k in range (10,21,2):     # Counts from 10-20 by 2s
   print(k, end = " ")
```

Repetition of multiple program statements works fine as long as proper, consistent indentation is used.

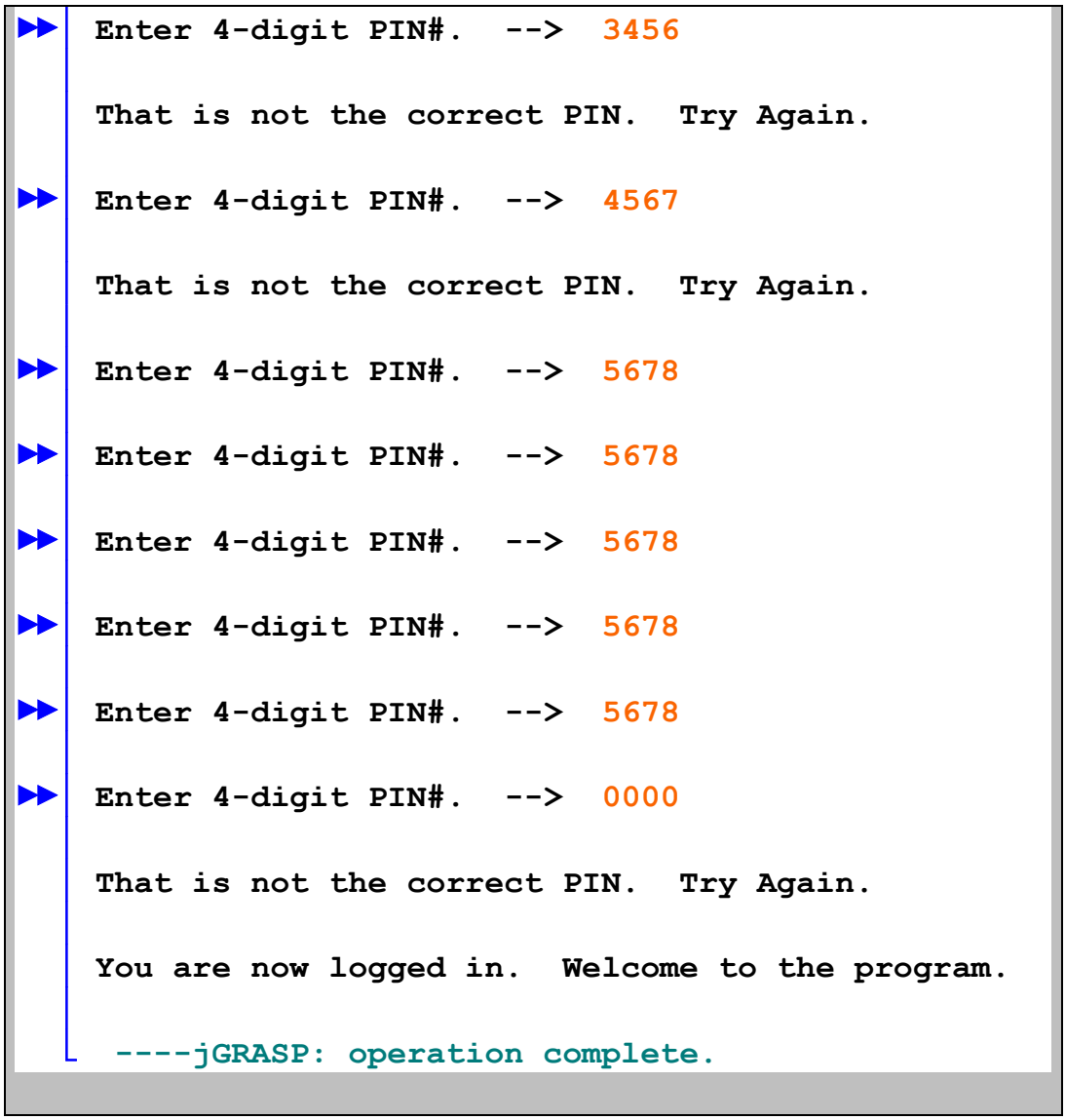
8.3 Conditional Repetition

The **for** loop is designed for *fixed* repetition. *Fixed* repetition means you know – ahead of time – how many times you want the loop to repeat. Program **Repetition09.py**, in Figure 8.9, is designed to show you that *fixed* repetition is not always possible. The program wants a user to enter a PIN (Personal Identification Number). A user should not be given access to the program unless they know the correct PIN. In this program, the correct PIN is **5678**. When the program is executed, you should notice the output is… a little strange.

Figure 8.9

```
 1 # Repetition09.py
 2 # This program is supposed to keep repeating until
 3 # a correct PIN of 5678 is entered.
 4 # The program does not work because the <for>
 5 # loop is used at a time that is not appropriate.
 6 # The <for> loop is meant for "fixed" repetition.
 7 # Entering a PIN is an example of "conditional" repetition.
 8
 9
10 for k in range(10):
11     print()
12     pin = input("Enter 4-digit PIN#.  -->  ")
13     if pin != "5678":
14         print("\nThat is not the correct PIN.  Try Again.")
15
16 print("\nYou are now logged in.  Welcome to the program.")
17
```

```
   ----jGRASP exec: python Repetition09.py

▶▶ Enter 4-digit PIN#.  -->  1234

   That is not the correct PIN.  Try Again.

▶▶ Enter 4-digit PIN#.  -->  2345

   That is not the correct PIN.  Try Again.
```



NOTE: The output of this program might cause a little confusion.

OK, this output should look seriously messed up. At first it is fine. The first 4 attempts at entering the password are incorrect. The user is told they did not enter the correct pin and to try again. On the 5th try something weird happens. The user enters the correct PIN of 5678. Since the password is correct, there is no message saying, "This is not the correct PIN." The strange part is the user has to enter the PIN again, and again and again and AGAIN and **AGAIN!** Even though the PIN was entered correctly 5 times, the user is still not permitted access. What is even weirder is that on the 10th attempt, when the user enters an incorrect password, the user is given access.

What causes this problem is the fact that the wrong type of loop was used. We used a **for** loop. This particular *fixed* repetition had the statement:

```
for k in range(10):
```

This is a loop that will repeat 10 times no matter what. It does not matter what PINs were entered. This program will ask for a PIN 10 times, and then give you access to the program, regardless of the PIN values. The purpose of the previous program was to demonstrate that *fixed* repetition is not always appropriate.

There are many situations where you know that something has to be repeated, but you do not know how many times. The entry of a password can be correct the first time or it may take many repeated attempts before the correct password is entered. Another situation may be based on time, which means that somebody who uses the computer slowly gets few repetitions and a fast operator gets many repetitions. Video games frequently repeat the same scene, but this repetition is usually not based on a fixed amount. Video games rely more on number of lives left. This is not always a simple "count down" because bonus lives can be earned. I think you get the picture. Life is full of situations where something is repeated, but it is not repeated a known number of times. Instead, it is repeated until something happens. We can also say it is repeated until some *condition* is met. This is called *conditional* repetition.

The while Loop

Program **Repetition10.py**, in Figure 8.10, introduces the **while** loop. This loop is a *conditional* loop. Like earlier selection control structures from the previous chapter, you will note that the *condition* is used. In this program the "condition" is that the PIN is correct. Specifically, the value of variable **pin** must be **5678**. Note that the condition actually says the opposite. Consider this. Do you want to make the user re-enter the PIN when he/she enters it correctly or incorrectly? It is when the PIN is incorrect that the loop needs to repeat. This is why the *condition* is **pin != 5678**. Now when you execute the program, it should behave properly. You are "welcomed" to the program once you enter the correct PIN, but not a moment before.

Figure 8.10

```
1 # Repetition10.py
2 # This program fixes the problem of the previous program
3 # by using a <while> loop.  Now the loop will stop when
4 # the correct PIN of 5678 is entered.
```

```
 5
 6
 7 pin = ""
 8 while pin != "5678":
 9    print()
10    pin = input("Enter 4-digit PIN#.  -->  ")
11    if pin != "5678":
12       print("\nThat is not the correct PIN.  Try Again.")
13
14 print("\nYou are now logged in.  Welcome to the program.")
15
```

```
 ----jGRASP exec: python Repetition10.py

Enter 4-digit PIN#.  -->  1234

That is not the correct PIN.  Try Again.

Enter 4-digit PIN#.  -->  2345

That is not the correct PIN.  Try Again.

Enter 4-digit PIN#.  -->  3456

That is not the correct PIN.  Try Again.

Enter 4-digit PIN#.  -->  4567

That is not the correct PIN.  Try Again.

Enter 4-digit PIN#.  -->  5678

You are now logged in.  Welcome to the program.

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

Now the output makes sense. On the first 4 attempts, when the **pin** is entered incorrectly, the user is asked to "Try Again." On the 5th attempt, when the correct **pin** of **5678** is entered, the loop stops as it should and the user is granted access.

Conditional Repetition

General Syntax:

initialize condition variable
while condition is True:
    execute program statement(s)

Specific Example:

```
password = ""
while password != "Qwerty2018":
   password = input("Enter password.  -->  ")
   if password != "Qwerty2018":
      print("Wrong password. Please re-enter")
print("Welcome.")
```

Fixed Repetition vs. Conditional Repetition

Fixed Repetition describes a situation where you know – ahead of time – how many times you want the loop to repeat.

An example would be drawing exactly 100 circles on the screen.

The command for fixed repetition is **for**.

Conditional Repetition describes a situation where you do NOT know how many times the loop will repeat.

The loop has to repeat until some *condition* is met.

An example would be entering a password.

The command for conditional repetition is **while**.

8.4 Nested Control Structures

In Chapter VII, there was a section called "Nested Selection" had examples of placing one selection structure inside another – similar to Russian nesting dolls. The next few examples will show examples placing one repetition control structure inside another. It should come as no surprise that this is called *Nested Repetition*.

You may wonder why the title of this section is not "Nested Repetition". The reason is that title is too restrictive for the examples that we will see. Yes, *Nested Selection* is placing one selection control structure in another and *Nested Repetition* is placing one repetition control structure in another. But turn back a couple pages and look at program **Repetition09.py**, in Figure 8.9, again. This program has a selection control structure inside a repetition control structure. Later, you will see an example of repetition control structures inside a selection control structure. While neither of these are "Nested Selection" or "Nested Repetition" they are examples of something more general. They are *Nested Control Structures*. When you place any control structure inside any other control structure, you have a *Nested Control Structure*. *Nested Selection* and *Nested Repetition* are just two specific examples of *Nested Control Structures*.

Program **Nested01.py**, in Figure 8.11, is our first example of "nested repetition". The program is not very clever. The inner loop repeats **4** times and displays the word "**Hello**". The outer loop repeats **3** times. The end result is "**Hello**" is displayed 12 times.

Figure 8.11

```
 1 # Nested01.java
 2 # This program demonstrates "Nested Repetition"
 3 # which is one type of "Nested Control Structure".
 4 # Since the outer loop repeats 3 times and the
 5 # inner loop repeats 4 times, the word "Hello"
 6 # is displayed 3 * 4 or 12 times.
 7
 8
 9 print()
10
11 for j in range(3):
12     for k in range(4):
13         print("Hello")
14
```

```
----jGRASP exec: python Nested01.py

Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello

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

The next program example, **Nested02.py**, in Figure 8.12, demonstrates the same nested loop structure as the previous program. The difference is, instead of displaying the word "**Hello**", the program displays the value of both *loop counters* or LCVs. Do both of these counters "count" at the same time? This program example helps you to see just what these two loops are doing. Both the outer loop counter, **j**, and the inner loop counter, **k**, are displayed. When you look at the output, you see that the inner loop counter counts much faster that the outer loop counter. You should notice that the inner loop must complete all of its repetitions before the outer loop can simply count to the next value.

Figure 8.12

```
1 # Nested02.java
2 # This program displays the value of the counters
3 # for both loops.  Note that the inner loop counts
4 # much faster than the outer loop.
5
6
7 print()
8
```

```
 9 for j in range(3):
10     for k in range(4):
11         print(j,k)
12
```

```
    ----jGRASP exec: python Nested02.py

 0 0
 0 1
 0 2
 0 3
 1 0
 1 1
 1 2
 1 3
 2 0
 2 1
 2 2
 2 3

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

Program **Nested03.py**, in Figure 8.13, is a nice example of a practical use of nested repetition. In this program example, a multiplication table in created with **15** rows and **15** columns. For now, the columns will not line up nicely. That will be fixed later. Note the two **print** commands on lines 12 and 13. The print command on line 12 is inside the *inner loop* and displays all of the multiples of a specific number on one row. On line 13, the other **print** command is indented differently. This **print** command is not in the *inner loop*. It is just in the *outer loop*. After the *inner loop*, displays all of the multiples of a specific number on one row, this **print** command forces the computer to go to the beginning if the next line so it is ready for the next row.

Figure 8.13

```
 1 # Nested03.java
 2 # This program displays a times table
 3 # that goes from 1 * 1 to 15 * 15.
 4 # In this program, the table does not
 5 # line up properly.
 6
 7
 8 print()
 9
10 for r in range(1,16):
11    for c in range(1,16):
12       print(r * c, end = " ")
13    print()
14
```

```
 ----jGRASP exec: python Nested03.py

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45
4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75
6 12 18 24 30 36 42 48 54 60 66 72 78 84 90
7 14 21 28 35 42 49 56 63 70 77 84 91 98 105
8 16 24 32 40 48 56 64 72 80 88 96 104 112 120
9 18 27 36 45 54 63 72 81 90 99 108 117 126 135
10 20 30 40 50 60 70 80 90 100 110 120 130 140 150
11 22 33 44 55 66 77 88 99 110 121 132 143 154 165
12 24 36 48 60 72 84 96 108 120 132 144 156 168 180
13 26 39 52 65 78 91 104 117 130 143 156 169 182 195
14 28 42 56 70 84 98 112 126 140 154 168 182 196 210
15 30 45 60 75 90 105 120 135 150 165 180 195 210 225

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

The reason the output does not line up properly in the output is that it contains a mixture of 1-digit, 2 digit and 3-digit numbers. This issue is addressed in the next program example.

Program **Nested04.py**, in Figure 8.14, fixes the lining up issue of the previous program by using a little *number formatting*. If you are not sure what "number formatting" is, please review the last section of Chapter VII.

Figure 8.14

```
 1 # Nested04.java
 2 # This program displays a better times table
 3 # where everything lines up properly by using
 4 # the <format> command.
 5
 6
 7 print()
 8
 9 for r in range(1,16):
10    for c in range(1,16):
11       print("{:3}".format(r * c), end = " ")
12    print()
13
```

```
----jGRASP exec: python Nested04.py

  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  2   4   6   8  10  12  14  16  18  20  22  24  26  28  30
  3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
  4   8  12  16  20  24  28  32  36  40  44  48  52  56  60
  5  10  15  20  25  30  35  40  45  50  55  60  65  70  75
  6  12  18  24  30  36  42  48  54  60  66  72  78  84  90
  7  14  21  28  35  42  49  56  63  70  77  84  91  98 105
  8  16  24  32  40  48  56  64  72  80  88  96 104 112 120
  9  18  27  36  45  54  63  72  81  90  99 108 117 126 135
 10  20  30  40  50  60  70  80  90 100 110 120 130 140 150
 11  22  33  44  55  66  77  88  99 110 121 132 143 154 165
 12  24  36  48  60  72  84  96 108 120 132 144 156 168 180
 13  26  39  52  65  78  91 104 117 130 143 156 169 182 195
 14  28  42  56  70  84  98 112 126 140 154 168 182 196 210
 15  30  45  60  75  90 105 120 135 150 165 180 195 210 225

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

Program **Nested05.py**, in Figure 8.15, repeats program **Selection12.py** from Chapter VII. When we look at this program, we see that it is another example of *Nested Control Structures*, specifically *Nested Selection*. One small issue with this program is that is needs to be run multiple times to interview multiple students.

Figure 8.15

```
 1 # Nested05.py
 2 # This program repeats Selection12.py from Chapter 7.
 3 # It is another example of "Nested Control Structures",
 4 # in this case, "Nested Selection".
 5
 6
 7 print()
 8 sat = eval(input("Enter SAT score  -->  "))
 9 print()
10
11 if sat >= 1100:
12    print("You are admitted.")
13    print("Orientation will start in June.")
14    print()
15    income = eval(input("Enter your family income  -->  "))
16    print()
17    if income < 20000:
18       print("You qualify for financial aid.")
19    else:
20       print("You do not qualify for financial aid.")
21 else:
22    print("You are not admitted.")
23    print("Please try again when your SAT improves.")
```

```
  ----jGRASP exec: python Nested05.py

▶▶ Enter SAT score  -->  1300

   You are admitted.
   Orientation will start in June.

▶▶ Enter your family income  -->  19000

   You qualify for financial aid.

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

```
----jGRASP exec: python Nested05.py

Enter SAT score  -->  1500

You are admitted.
Orientation will start in June.

Enter your family income  -->  99000

You do not qualify for financial aid.

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

```
----jGRASP exec: python Nested05.py

Enter SAT score  -->  700

You are not admitted.
Please try again when your SAT improves.

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

The next few programs explore different ways to address the issue of having to execute this program again and again when interviewing multiple students.

Program **Nested06.py**, in Figure 8.16, essentially nests the previous program inside a **for** loop that repeats **5** times. This allows the interviewer to interview 5 students with a single execute. Each student gives his/her SAT score and family income and then receives a response. At the end of the **for** loop I added a separator (line of dashes) to separate the information from each student.
While this may solve the problem of the previous program, it does introduce another. What if you need to interview more than 5 students? What if you need to interview less? This program only works when you are interviewing exactly 5 students.

This program also demonstrates a couple other things that are significant. First, it shows that you can have more than 2 levels of "nesting". In this case, we have *selection* nested inside *selection* nested inside *repetition*. Second, it shows that in order for this to work, you need to be even more careful with proper, consistent indentation.

Figure 8.16

```
 1 # Nested06.py
 2 # This program demonstrates that control structures
 3 # can be nested with more than 2 levels.
 4 # This program is actually the entire previous
 5 # program nested inside a loop that repeats 5 times.
 6 # NOTE: As you have more and more levels of nesting
 7 #       indentation becomes more and more important.
 8 # ALSO: This program does have an issue in that it
 9 #       basically assumes you will always interview
10 #       exactly 5 students.
11
12 for k in range(5):
13    print()
14    sat = eval(input("Enter SAT score  -->  "))
15    print()
16    if sat >= 1100:
17       print("You are admitted.")
18       print("Orientation will start in June.")
19       print()
20       income = eval(input("Enter your family income  -->  "))
21       print()
22       if income < 20000:
23          print("You qualify for financial aid.")
24       else:
25          print("You do not qualify for financial aid.")
26    else:
27       print("You are not admitted.")
28       print("Please try again when your SAT improves.")
29    print("\n-----------------------------------")
30
```

```
----jGRASP exec: python Nested06.py

Enter SAT score  -->  900

You are not admitted.
Please try again when your SAT improves.
```

```
-----------------------------------

Enter SAT score  -->  1000

You are not admitted.
Please try again when your SAT improves.

-----------------------------------

Enter SAT score  -->  1099

You are not admitted.
Please try again when your SAT improves.

-----------------------------------

Enter SAT score  -->  1100

You are admitted.
Orientation will start in June.

Enter your family income  -->  18000

You qualify for financial aid.

-----------------------------------

Enter SAT score  -->  1200

You are admitted.
Orientation will start in June.

Enter your family income  -->  150000

You do not qualify for financial aid.

-----------------------------------

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

Program **Nested07.py**, in Figure 8.17, improves on the previous program by making it more flexible. Instead of always repeating **5** times, the program will start by asking you how many students you wish to interview. The number you type is stored in the variable **numStudents**. Note that **numStudents** is now used in the **range** command instead of a constant number. After the **for** loop heading, the rest of the program is identical to the previous program.

Figure 8.17

```
 1 # Nested07.py
 2 # This program is very similar to the previous
 3 # program.  The different is that it begins with
 4 # an input statement that allows the interviewer
 5 # to enter the number of students that he/she
 6 # needs to interview.
 7
 8 print()
 9 numStudents = eval(input("How many students do you need
to interview?  -->  "))
10
11 for k in range(numStudents):
12    print()
13    sat = eval(input("Enter SAT score  -->  "))
14    print()
15    if sat >= 1100:
16       print("You are admitted.")
17       print("Orientation will start in June.")
18       print()
19       income = eval(input("Enter your family income  -->  "))
20       print()
21       if income < 20000:
22          print("You qualify for financial aid.")
23       else:
24          print("You do not qualify for financial aid.")
25    else:
26       print("You are not admitted.")
27       print("Please try again when your SAT improves.")
28    print("\n-----------------------------------")
29
```

```
   ----jGRASP exec: python Nested07.py

▶▶ How many students do you need to interview?  -->  3
```

```
Enter SAT score  -->  1300

You are admitted.
Orientation will start in June.

Enter your family income  -->  19000

You qualify for financial aid.

-------------------------------------

Enter SAT score  -->  1500

You are admitted.
Orientation will start in June.

Enter your family income  -->  99000

You do not qualify for financial aid.

-------------------------------------

Enter SAT score  -->  700

You are not admitted.
Please try again when your SAT improves.

-------------------------------------

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

This program works fine, but there is one more potential issue that we have not addressed. What if you have a room full of people and there are too many to count? A more practical program would keep going until you are ready to stop. At the conclusion of each interview, program **Nested08.py**, shown in Figure 8.18, will ask if you wish to interview another student. A response of '**Y**' will make the program repeat. Entering anything else, even a lowercase '**y**' will terminate the program. This works because a *conditional* **while** used is used instead of a *fixed* **for** loop.

Figure 8.18

```
1  # Nested08.py
2  # This program fixes the issue of the previous program.
3  # Now everything is inside a <while> loop.
4  # At the conclusion of each interview the user has
5  # the option to repeat the program.
6  # The <while> loop makes the program repeat as long
7  # as the user responds with a capital 'Y'.
8
9  response = 'Y';
10
11 while response == 'Y':  # Note: Only capital 'Y' will make the loop repeat.
12    print()
13    sat = eval(input("Enter SAT score  -->  "))
14    print()
15    if sat >= 1100:
16       print("You are admitted.")
17       print("Orientation will start in June.")
18       print()
19       income = eval(input("Enter your family income  -->  "))
20       print()
21       if income < 20000:
22          print("You qualify for financial aid.")
23       else:
24          print("You do not qualify for financial aid.")
25    else:
26       print("You are not admitted.")
27       print("Please try again when your SAT improves.")
28    print()
29    response = input("Do you want to interview another student?  {Y/N}  -->  ")
30
```

```
----jGRASP exec: python Nested08.py

Enter SAT score  -->  1300

You are admitted.
Orientation will start in June.
```

```
Enter your family income  -->  19000

You qualify for financial aid.

Do you want to interview another student?  {Y/N}  -->  Y

Enter SAT score  -->  1500

You are admitted.
Orientation will start in June.

Enter your family income  -->  99000

You do not qualify for financial aid.

Do you want to interview another student?  {Y/N}  -->  Y

Enter SAT score  -->  700

You are not admitted.
Please try again when your SAT improves.

Do you want to interview another student?  {Y/N}  -->  N

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

You may wonder why the program will only repeat if a capital '**Y**' is entered. Should it not also repeat for a lowercase '**y**'? The answer is "Yes, it should." However, this requires a more complicated type of *condition*. You will learn how to do this in Chapter XI.

We have seen several examples of *selection* structures nested inside *repetition* structures. Program **Nested09.py**, in Figure 8.19, shows the reverse: *repetition* structures nested inside a *selection* structure. The program first enters an integer. The **if** statement says **if number % 2 == 0**. *If the remainder of number divided by 2 equals 0.* What does that mean? If we divide by 2 and have nothing left over, that means we have an *even* number. So, the condition in the **if** statement essentially determines if **number** is *even*. If this condition is **true**, the word "**EVEN**" is displayed **number** times. If this condition is **false**, the number is *odd* and the word "**ODD**" is displayed **number** times.

Figure 8.19

```
 1 # Nested09.py
 2 # This program demonstrates that repetition can
 3 # be nested inside a selection.
 4 # In truth, ANY control structure can be nested
 5 # inside ANY other control structure.
 6 # The program also shows how to determine if a
 7 # number is even or odd.
 8
 9
10 print()
11 stop = eval(input("Enter a number between 1 and 15.  -->  "))
12 print()
13
14 if (stop % 2 == 0):  # if stop is even
15    for k in range(stop):
16       print("EVEN",end = " ")
17 else:                # if stop is odd
18    for k in range(stop):
19       print("ODD",end = " ")
20
21 print()
```

```
   ----jGRASP exec: python Nested09.py

▶▶ Enter a number between 1 and 15.  -->  10

   EVEN EVEN EVEN EVEN EVEN EVEN EVEN EVEN EVEN EVEN

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

```
   ----jGRASP exec: python Nested09.py

▶▶ Enter a number between 1 and 15.  -->  13

   ODD ODD ODD ODD ODD ODD ODD ODD ODD ODD ODD ODD ODD

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

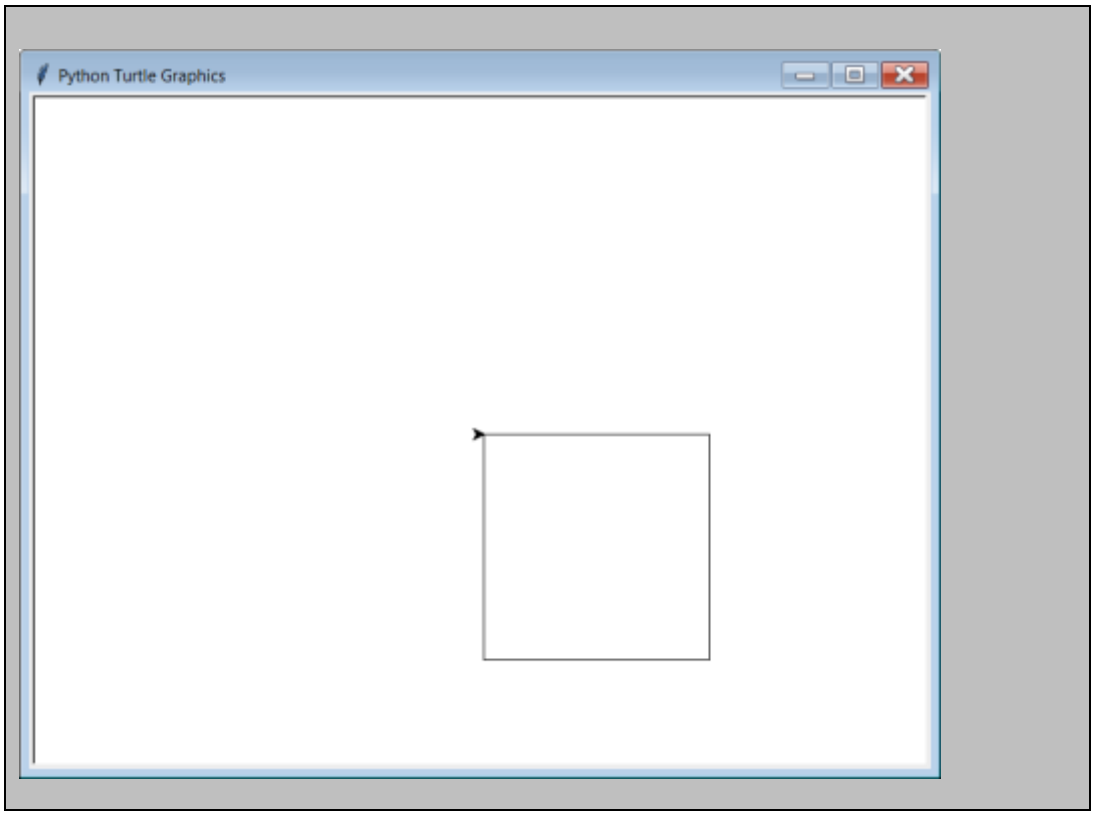
8.5 Using Repetition with Turtle Graphics

You have already learned two ways to create graphics output in Python. One is to use *Turtle Graphics* from the built-in **turtle** library. The other is to use more *Traditional Graphics* from Mr. Schram's **Graphics** library. You may be thinking that graphics programs are simple. You need a circle, use the **drawCircle** command. You need a solid rectangle, use the **fillRectangle** command. It needs to be understood that there is a lot more to creating graphics programs that simply choosing the right command for the right purpose. You can also perform some neat tricks by using control structures, specifically the **for** loop. Repetition combined with graphics procedure calls can have some very nice visual effects.

The program examples in this section will examine the effects of combining repetition with *Turtle Graphics*. Figure 8.20 shows program **RepetitionWithGraphics01.py**, which repeats program **TurtleGraphics08.py** from Chapter V. The program draws a square, but it does so in an inefficient manner by repeating the exact same program statements 4 times.

Figure 8.20

```
 1 # RepetitionWithGraphics01.py
 2 # This program repeats TurtleGraphics08.py
 3 # to demonstrate an inefficient way to draw
 4 # a square.
 5
 6
 7 from turtle import *
 8
 9 setup(800,600)
10
11 forward(200)
12 right(90)
13 forward(200)
14 right(90)
15 forward(200)
16 right(90)
17 forward(200)
18 right(90)
19
20 update()
21 done()
```



Program **RepetitionWithGraphics02.py**, in figure 8.21, draws the exact same square as the previous program, but improves the efficiency by adding a **for** loop.

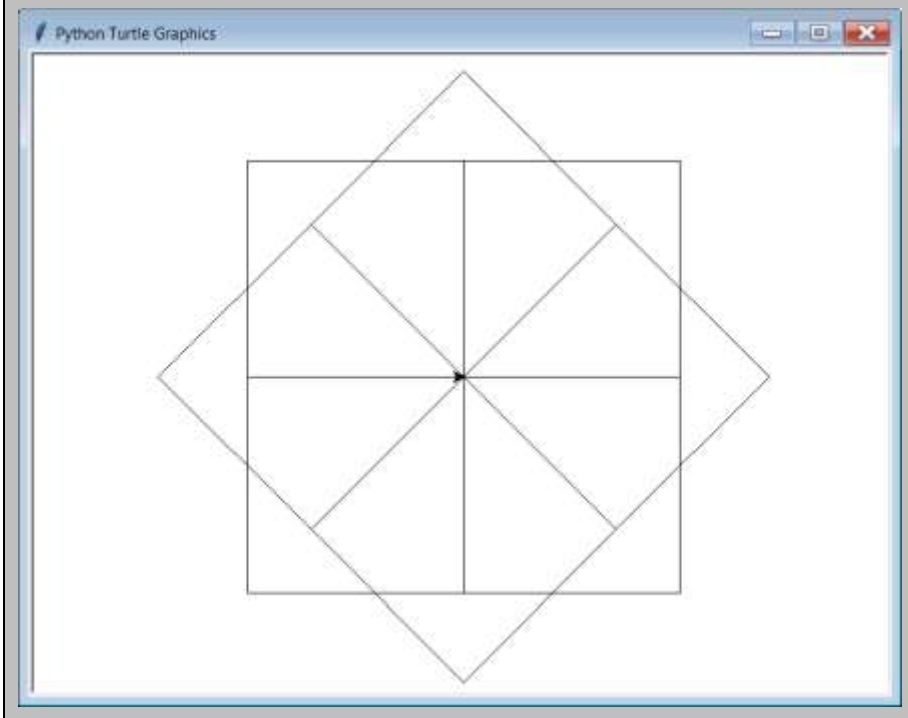
Figure 8.21

```
1 # RepetitionWithGraphics02.py
2 # This program draws the same square as the previous
3 # program, but is more efficient because it uses a
4 # <for> loop to create the square.
5
6
7 from turtle import *
8
9 setup(800,600)
10
11 for k in range(4):
12     forward(200)
13     right(90)
14
15 update()
16 done()
```

Program **RepetitionWithGraphics03.py**, in figure 8.22, *Nested Repetition* can be used to create special designs. Essentially, the "square loop" from the previous program is nested inside another **for** loop. After each square is drawn, the turtle turns **left 45** degrees. The output shows the resulting design.

Figure 8.22

```
# RepetitionWithGraphics03.py
# This program takes the "square loop" from the
# previous program and "nests" it inside another
# <for> loop to create a special design.


from turtle import *

setup(800,600)

for j in range(8):
    for k in range(4):
        forward(200)
        right(90)
    left(45)

update()
done()
```

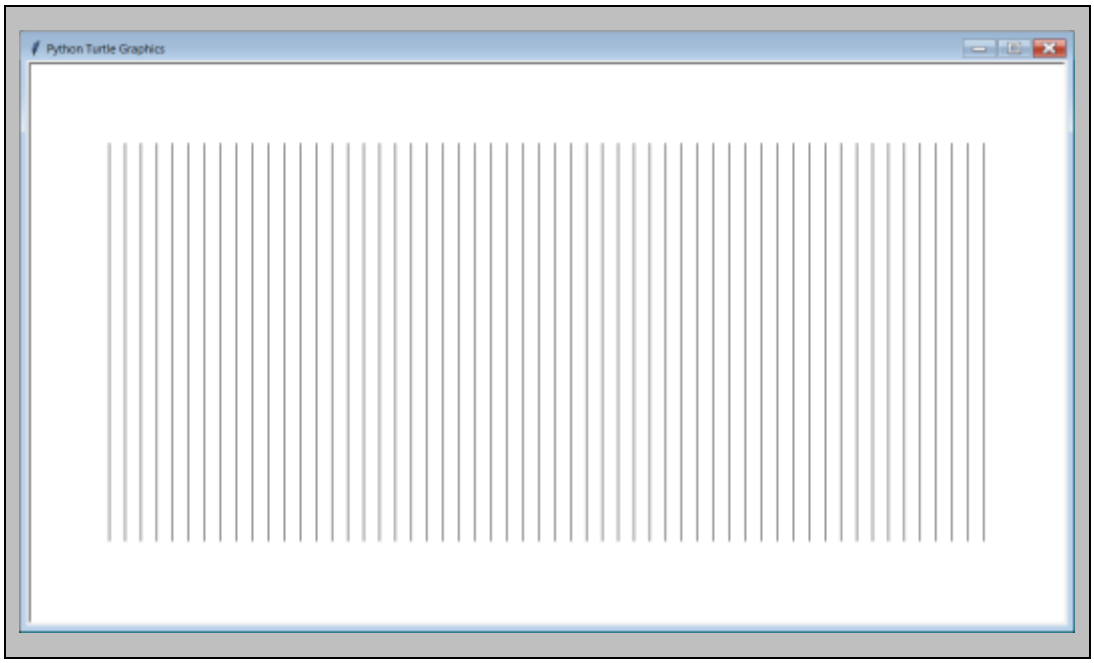


8.6 Using Repetition with Traditional Graphics

The program examples in this section will use *Traditional Graphics*. Program **RepetitionWithGraphics04.java**, in Figure 8.23, draws many vertical lines. There are 4 separate integer variables used to draw each line. These are **x1**, **y1**, **x2** and **y2**. At the beginning of the program these 4 variables are initialized with the values necessary to draw the first line. The lines are *parallel* and *vertical* because both *x* values are always the same. Note that both variables are initialized to **100**. In the **for** loop the same value, in this case **20**, is added to both **x1** and **x2**. This causes every new line to be drawn **20** pixels to the right of the previous line. Note that **y1** and **y2** values, which control the top-down values, never change. They are fixed at **100** and **600** respectively. Also, the **56** in the **range** command specifies that we want **56** lines drawn. With the help of a **for** loop you can draw as many lines as you wish with just a single **drawLine** command.

Figure 8.23

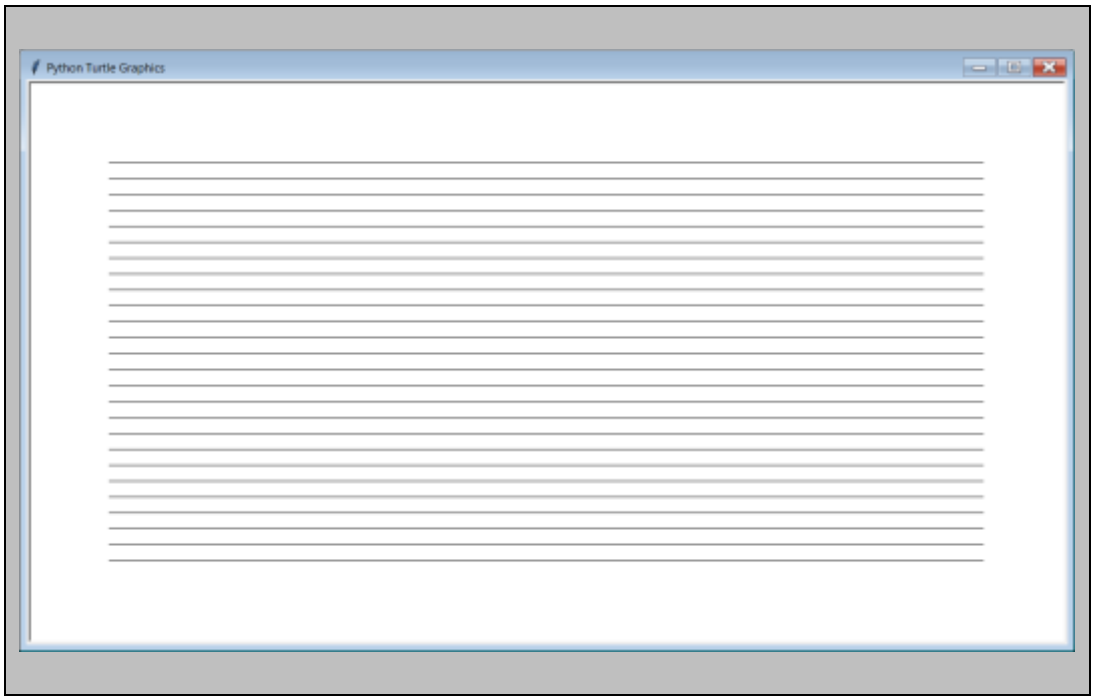
```
# RepetitionWithGraphics04.py
# This program shows how a <for> loop can also be
# used with traditional graphics. This program draws
# vertical lines because <x1> and <x2> are always equal.


from Graphics import *

beginGrfx(1300,700)

x1 = 100
y1 = 100
x2 = 100
y2 = 600

for k in range(56):
    drawLine(x1,y1,x2,y2)
    x1 += 20
    x2 += 20

endGrfx()
```



Program **RepetitionWithGraphics05.py**, in Figure 8.24, makes a few small changes to the previous program to draw horizontal lines. This time, the values of **x1** and **x2** are fixed while the identical values of **y1** and **y2** increase in the **for** loop.

Figure 8.24

```
# RepetitionWithGraphics05.py
# This program draws horizontal lines because
# <y1> and <y2> are always equal.


from Graphics import *

beginGrfx(1300,700)

x1 = 100
y1 = 100
x2 = 1200
y2 = 100

for k in range(26):
   drawLine(x1,y1,x2,y2)
   y1 += 20
   y2 += 20

endGrfx()
```



In the graphics department, where **for** loops are combined with graphics commands, programs that draw horizontal lines and vertical lines are pretty comfortable. One notch higher on the complexity ladder would be programs that draw diagonal lines.

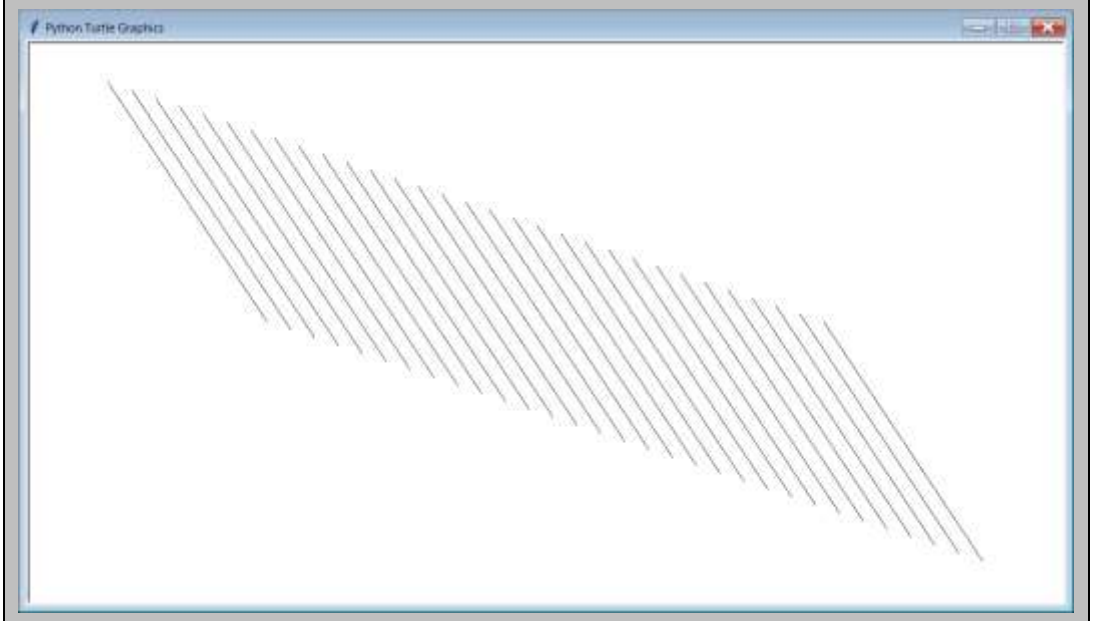
Program **RepetitionWithGraphics06.py**, in Figure 8.25, shows a different approach from the two previous programs. Before either the *x* values were altered or the *y* values were altered. Now both are altered.

As with the previous 2 programs, the four coordinate values required to draw the first diagonal line are initialized before the **for** loop. After each line is drawn, each of the four values is changed in preparation for the next line. **30** is added to both *x* values. **10** is added to both *y* values. The result is that every line is **30** pixels *over* and **10** pixels *down* from the previous line.

Figure 8.25

```
1 # RepetitionWithGraphics06.py
2 # This program draws parallel diagonal lines
3 # and changes all 4 variables.
4
5
6 from Graphics import *
7
8 beginGrfx(1300,700)
9
```

```
10 x1 = 100
11 y1 = 50
12 x2 = 300
13 y2 = 350
14
15 for k in range(31):
16    drawLine(x1,y1,x2,y2)
17    x1 += 30
18    x2 += 30
19    y1 += 10
20    y2 += 10
21
22 endGrfx()
23
```

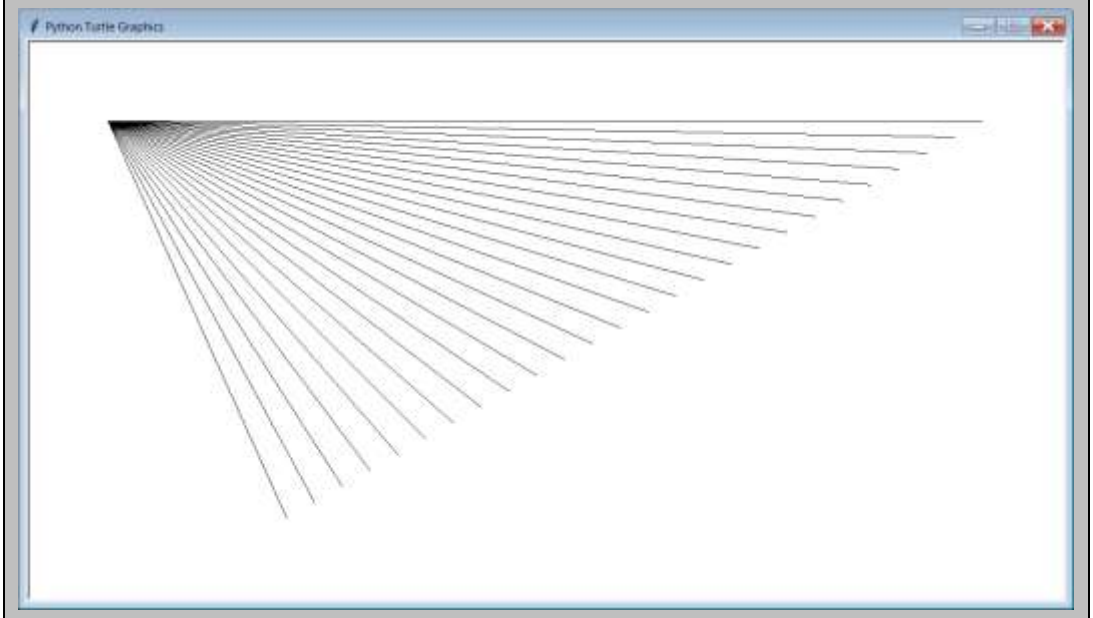


To really understand this program, go to lines 17-20 and change the **10**s and **30**s to other numbers. You can make all 4 numbers different. You can even make them negative. See what kind of designs you can come up with.

Program **RepetitionWithGraphics07.py**, in Figure 8.26, plays another twist with straight lines. This time it is not a set of *x* values or a set of *y* values that changes. Instead, the first coordinate (**x1,y1**) is fixed. In the second coordinate (**x2,y2**) both values change.

Figure 8.26

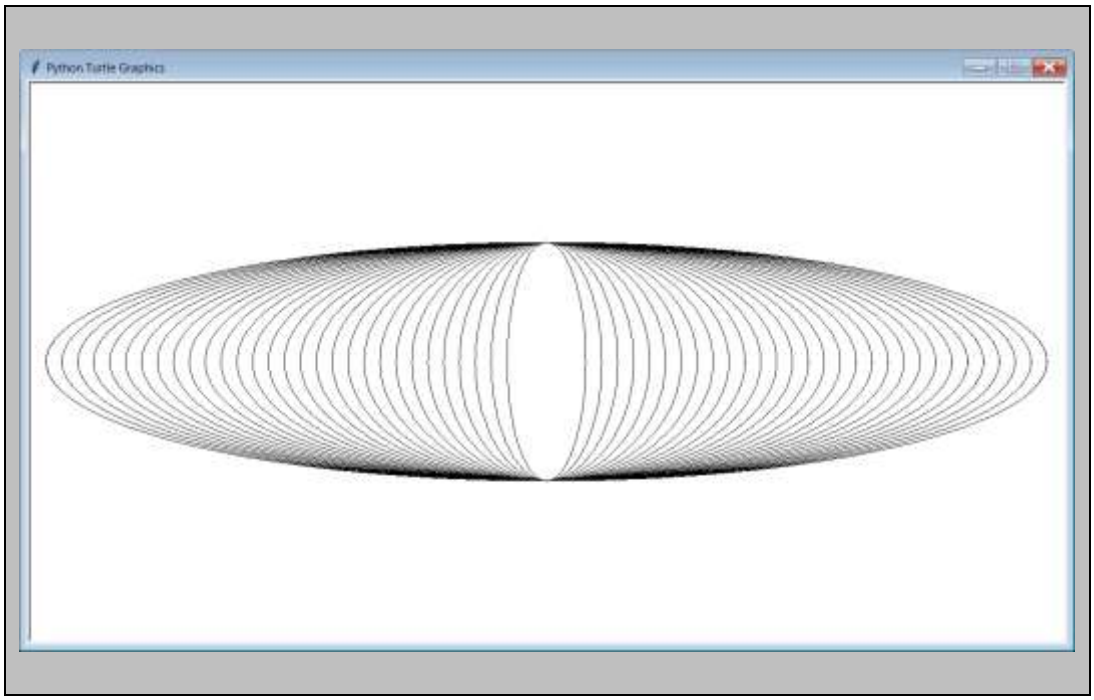
```
 1 # RepetitionWithGraphics07.py
 2 # This program demonstrates several lines with the same
 3 # starting point. In this case the (x1,y1) coordinate
 4 # stays fixed while the (x2,y2) point changes.
 5
 6
 7 from Graphics import *
 8
 9 beginGrfx(1300,700)
10
11 x1 = 100
12 y1 = 100
13 x2 = 1200
14 y2 = 100
15
16 for k in range(26):
17     drawLine(x1,y1,x2,y2)
18     x2 -= 35
19     y2 += 20
20
21 endGrfx()
22
```



Straight lines are nice, but repetition can also be done with points, circles, ovals, rectangles, stars, or any other shape. The only limit is your imagination. Program **RepetitionWithGraphics08.py**, in Figure 8.27, is another very small program. This one draws ovals. Like the previous programs, it sets up all of the variables before the **for** loop. The loop makes sure 3**0** ovals are drawn. If you look at the output you will see the ovals. The first oval is the one in the middle. Every time the loop is repeated only one value is changed. The variable **hr** (for *horizontal radius*) has **20** added to it. Nothing else is changed. This means every oval has the same center (**x,y**) coordinate, and the same *vertical radius*. Since the *horizontal radius* increase by **20** each time, each oval gets **40** pixels fatter. If you are wondering, *"Why 40, and not 20?"* Remember that the *diameter* of a circle/oval is *twice* its *radius*. If the radius increases by **20**, the diameter increases by **40**.

Figure 8.27

```
 1 # RepetitionWithGraphics08.py
 2 # This program demonstrates several ovals.
 3 # All of the ovals have the same center and vertical
 4 # radius.  The horizontal radius keeps growing.
 5 # NOTE: There may be a little delay in the display
 6 # of the output.
 7
 8
 9 from Graphics import *
10
11 beginGrfx(1300,700)
12
13 x = 650
14 y = 350
15 hr = 50
16 vr = 150
17
18 for k in range(30):
19     drawOval(x,y,hr,vr)
20     hr += 20
21
22 endGrfx()
23
```

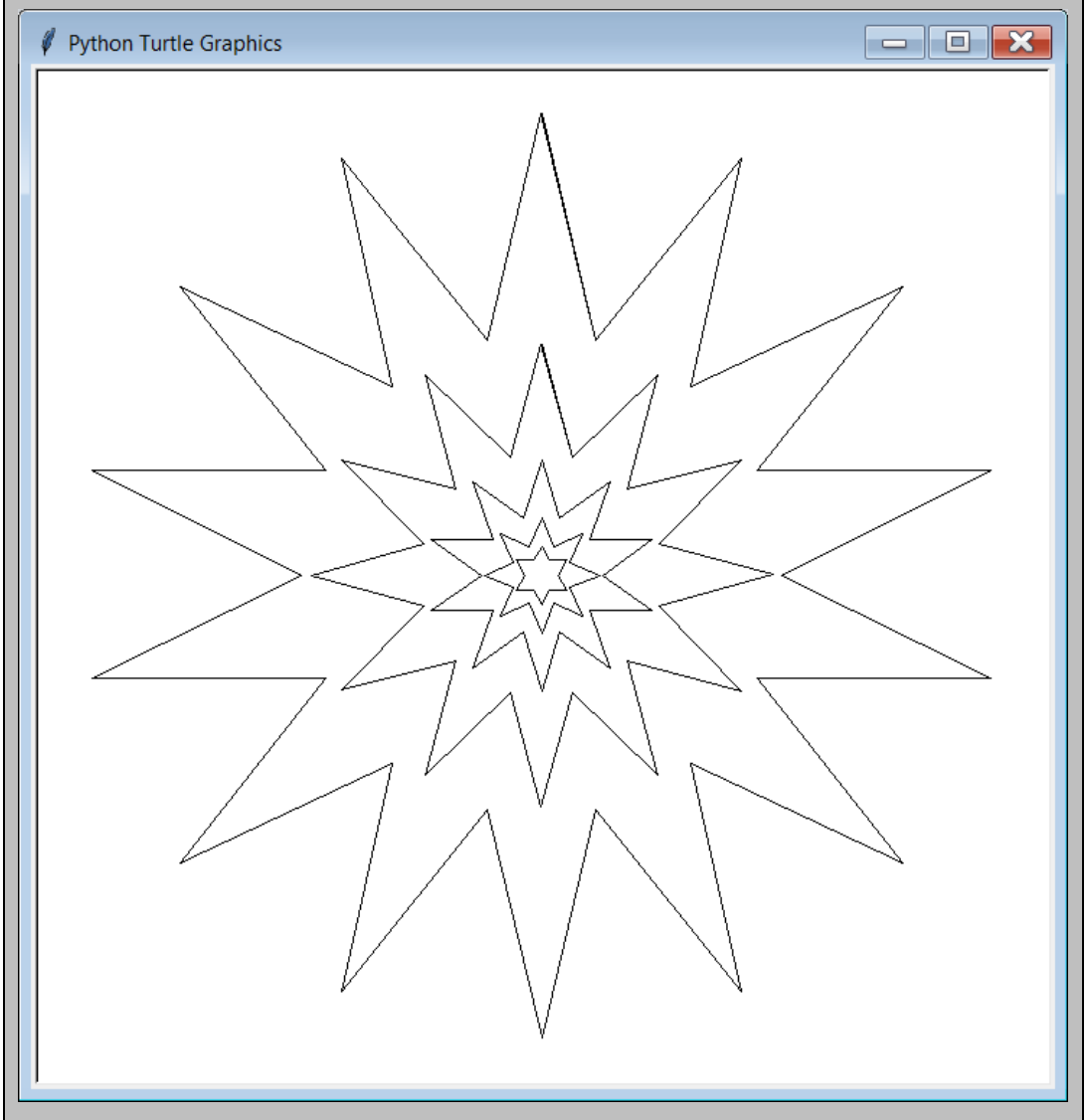


Program **RepetitionWithGraphics09.py**, shown in Figure 8.28, is similar to the previous program, but this one draws stars instead of ovals. The first star has **6** points and a radius of **20**. Every time through the loop, each new star has 2 additional points and double the radius of the previous star.

Figure 8.28

```
# RepetitionWithGraphics09.py
# This program demonstrates several concentric stars.
# Each new star has 2 more points and double the radius
# of the previous star.


from Graphics import *

beginGrfx(700,700)

x = 350
y = 350
r = 20
p = 6

```

```
16 for k in range(5):
17     drawStar(x,y,r,p)
18     p += 2
19     r *= 2
20
21 endGrfx()
22
```



The final program in this section, program **RepetitionWithGraphics10.py**, shown in Figure 8.29, also draws stars. The difference now is the stars are solid or "filled-in". They also now have different colors. To make this work, I am using the **setColor** procedure in a new way.

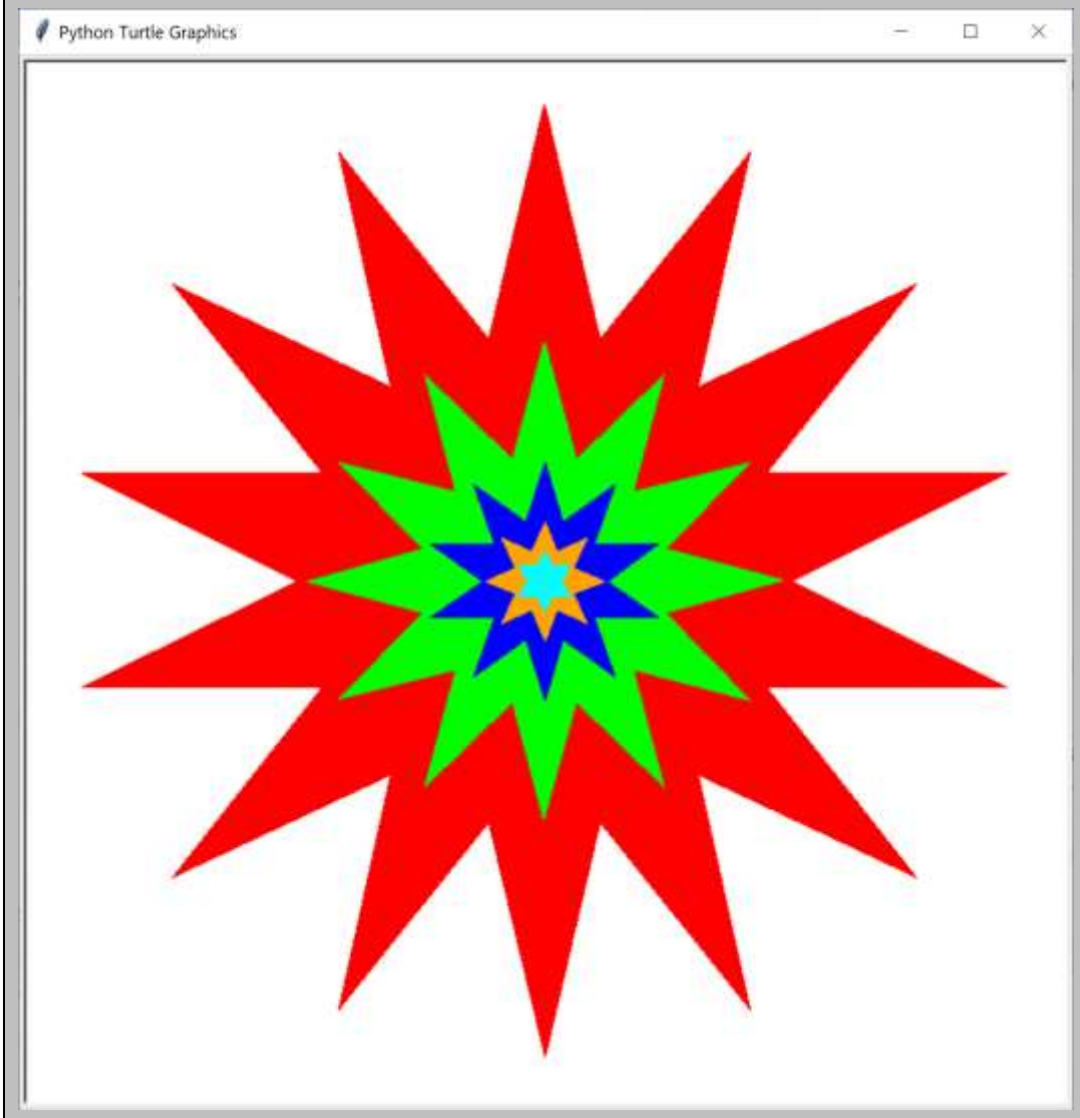
The first time you saw **setColor**, it was with a string literal argument. For example: `setColor("red")`. In the previous section, you saw **setColor** being used with 3 integer arguments which specify the amount of **red**, **green** and **blue** in a color. For example: `setColor(150, 100, 15)`. There is a third way to use **setColor** which is shown in this program. **setColor** can be used with a single integer parameter. This was created to provide a simple way to cycle through the primary colors in a loop. With this system `setColor(0)` gives you **black**. `setColor(1)` gives you **red**, `setColor(2)` gives you **green**, `setColor(3)` gives you **blue**, `setColor(4)` gives you **orange**, etc.

There is something else that you should notice about this program. It actually works in a reverse manner from the previous program. In the previous program, the first star had 6 points and a radius of 20. In the loop, 2 was added to the number of points and the radius was doubled until the last star which had 14 points and a radius of 320. For this program, with solid stars, this had to be reversed. You may remember the "Target" from Lab 6B. Now the first star has 14 points and a radius of 320. In the loop, 2 is subtracted from the number of points and the radius is chopped in half until the last star which has 6 points and a radius of 20.

Figure 8.29

```
# RepetitionWithGraphics10.py
# This program is similar to the previus program, but now
# the stars are filled in with solid colors. It also shows
# that <setColor> can be used with a single int parameter:
# 0=Black, 1=Red, 2=Green, 3=Blue, 4=Orange, 5=Cyan, etc.
# Note that the radius <r> and number of points <p> variables
# are counting in the opposite direction from before.


from Graphics import *

beginGrfx(700,700)

x = 350
y = 350
r = 320
```

```
17 p = 14
18 c = 1
19
20 for k in range(5):
21     setColor(c)
22     fillStar(x,y,r,p)
23     p -= 2
24     r //= 2
25     c += 1
26 endGrfx()
27
```



8.7 Creating Custom Colors

Python provides you with 140 different colors. Suppose that is not enough. Let us simply look at the color **blue**. Yes, there is **blue**, **AliceBlue**, **CadetBlue**, **CornflowerBlue**, **DarkBlue**, **DarkSlateBlue**, **DeepSkyBlue**, **DodgerBlue**, **LightBlue**, **LightSkyBlue**, **LightSteelBlue**, **MediumBlue**, **MediumSlateBlue**, **MidnightBlue**, **Navy**, **RoyalBlue**, **SlateBlue**, and **SteelBlue**… and I am not even listing all of the shades of **aqua**, **cyan**, and **turquoise**. But what if you still want more?

What if you want the exact shade of blue to match one of your school's colors? What if you are the romantic type and you want the exact shade of blue to match your girlfriend's eyes? Python can actually create over 16 million different colors, so any color you can think of is possible. It is like those hardware store commercials where someone brings in toy, points to it, and says, "Can you match this?" Then 10 minutes later he walks out of the hardware store with a gallon of paint that is the exact same color as the toy. Whether you work in the Paint Department of a hardware store, or are writing a graphics program in Computer Science class, the process is pretty much the same – at least logically. (We will not be mixing any actual paint.)

On the computer, you can create any color, by mixing certain amounts of the 3 primary colors: **red**, **green**, and **blue**. Usually when we say this, there are a couple students who have flashbacks to finger-painting in kindergarten. A certain *Barney* song starts playing in their head,

> *"When you mix **blue** and **yellow** it makes **green**…"*

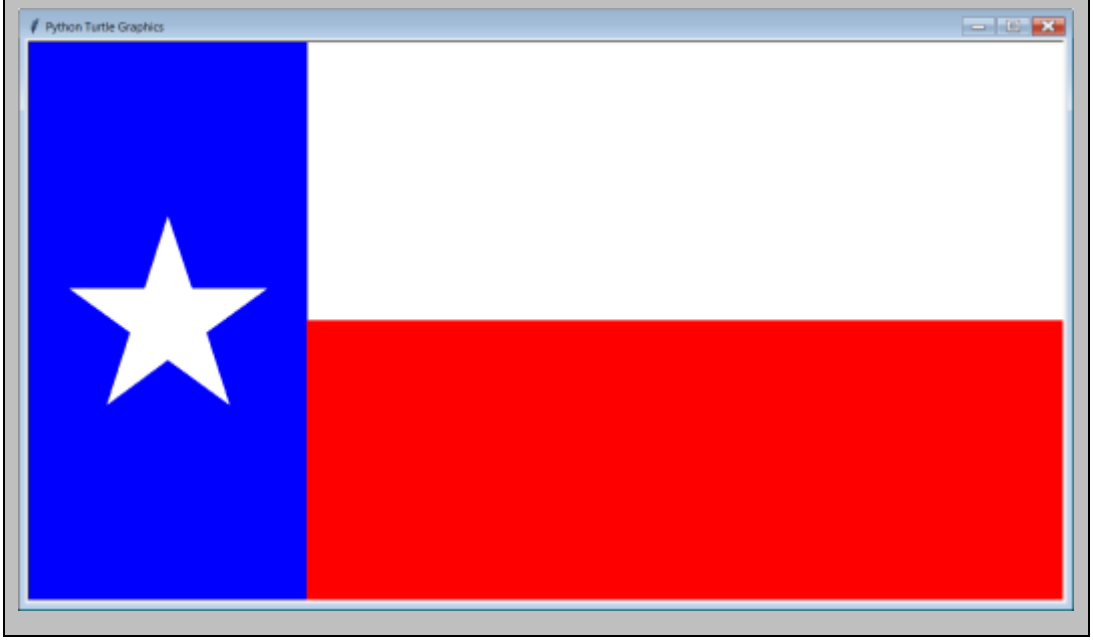
We are not saying that *Barney* was wrong. When you work with *paint* the primary colors are **red**, **yellow**, and **blue**; however, when you with *light* (as in the light that comes from your computer screen) the primary colors are **red**, **green**, and **blue**. We are right now not going to be concerned with the exact physics of how the computer can produce **yellow** by mixing **red**, **green**, and **blue**. If you are really curious about that, I suggest you ask your science teacher.

In order to create the new colors, we are going to use the **setColor** procedure of the **Graphics** library in a different way. Instead of using a string literal to specify the name of the color we want, we will use 3 integer arguments. The first is a number from between **0** and **255** which indicates how much **red** you want to use. The second is another number in the [0..255] range which indicates how much **green** you want to use. The third is yet another number in the [0..255] range which indicates how much **blue** you want to use. Make sure all 3 numbers are between **0** and **255** or things will simply not work!

Program **CustomColors01.py**, in Figure 8.30, starts things off by drawing the Texas Flag. Consider this. Given that the **Graphics** library has a **fillStar** procedure, is the Texas Flag not a really simple program? It is basically just 3 rectangles and a star. The colors **red**, **white**, and **blue** are built into Python so we should have no trouble.

Figure 8.30

```
# CustomColors01.py
# This program displays the Texas flag using the
# built-in colors for red, white and blue.  While
# this certainly looks like the Texas flag, it does
# not use the official shades of red and blue that
# are used in the Official Texas Flag.


from Graphics import *

beginGrfx(1300,700)

setColor("blue")
fillRectangle(0,0,350,700)
setColor("red")
fillRectangle(350,350,1300,700)
setColor("white")
fillStar(175,350,130,5)

endGrfx()
```



While the output of program **CustomColors01.py** looks a lot like the Texas Flag, it does not use the official shades of **red** and **blue** that are used in the Official Texas Flag. To accomplish this, we need to create our own *custom colors*. Program **CustomColors02.py**, in Figure 8.31, uses 3 integer arguments instead of a color name in each of its **setColor** procedure calls. The result is we now have the official colors of the Official Texas Flag.

Figure 8.31

```
1  # CustomColors02.py
2  # This program demonstrates the <setColor> procedure
3  # being used to "create" custom colors.  The program
4  # will draw the Official Texas Flag with the EXACT
5  # official shades of red and blue required.
6
7
8  from Graphics import *
9
10 beginGrfx(1300,700)
11
12 setColor(0,39,104)      # Texas Flag blue
13 fillRectangle(0,0,350,700)
14 setColor(190,10,47)     # Texas Flag red
15 fillRectangle(350,350,1300,700)
16 setColor(255,255,255)   # Three 255s = "white"
17 fillStar(175,350,130,5)
18
19 endGrfx()
```

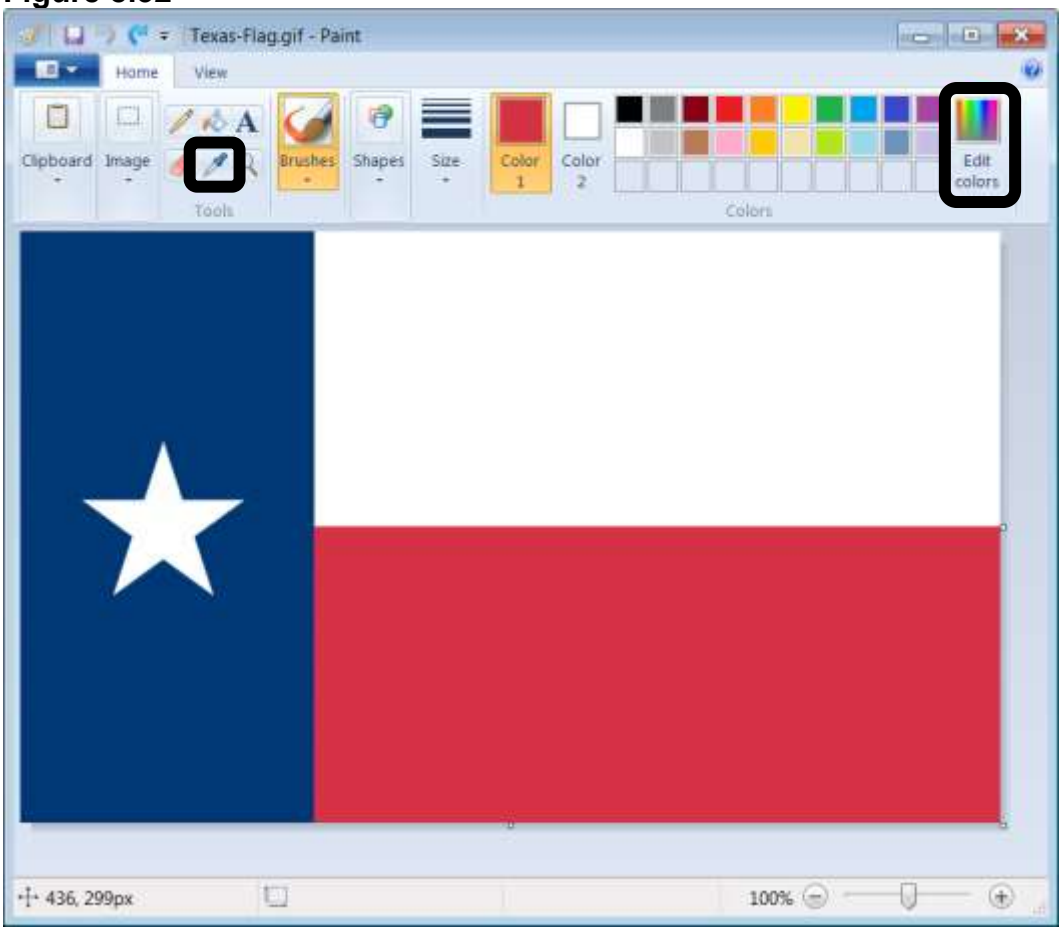
Where did you get the 3 numbers?

So one question that you might have after the previous program is where did I get the 3 integer values that I used to create "Texas Flag red" and "Texas Flag blue?" I mean, it is one thing to look at the command **setColor(0,39,104)** and know that "Texas Flag blue" has a red value of **0**, a green value of **39**, and a blue value of **104**. Since this is a "shade of blue" it makes sense that the "blue value" would be the largest. However, where did Mr. Schram gets these number in the first place? Well, here is what did:

1) I went to the official website of the state of Texas, http://www.texas.gov and I looked for an image of the Official State Flag.

2) I downloaded the image, **Texas-Flag.gif** and then loaded it into **MicroSoft Paint** as shown in Figure 8.32.

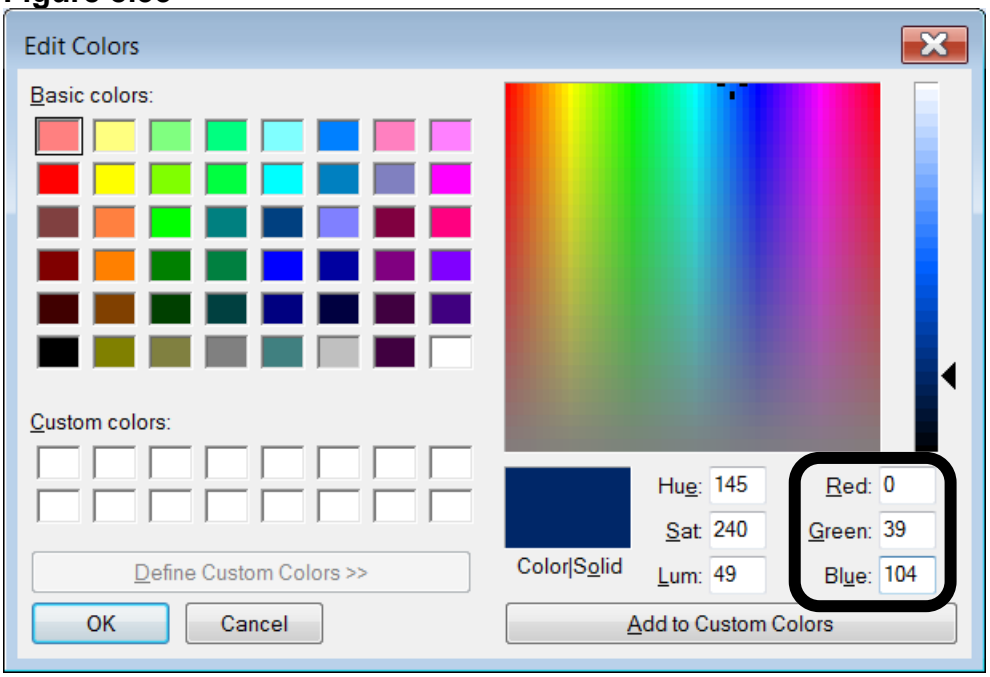
Figure 8.32



3) I selected the *eye-dropper* tool,  and clicked in the blue field of the Texas Flag.

4) I then clicked **Edit colors** and wrote down the values shown for **Red**, **Green** and **Blue** as see in Figure 8.33.

Figure 8.33



5) Using the values of **0,39,104** in my **setColor** command gave me the exact shade of "Texas Flag blue" that I wanted in my output.

6) I then repeated steps 3-5 to get the numbers for "Texas Flag red".

What if I have a Mac?

You can accomplish the same thing on a Mac. The steps are a bit different.

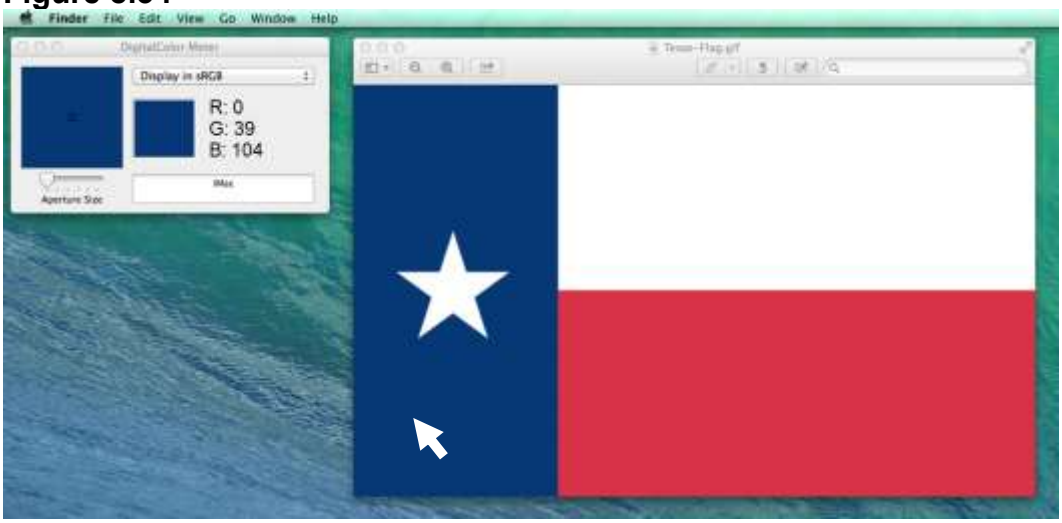
1) Once you have the image downloaded or saved on your computer, double-click it which will open a Preview window and make the image visible on the desktop.

2) Open **DigitalColor Meter**. (You can do a *search* for it.)

3) Make sure the **Display** setting says **Display in sRGB**.

4) Move your cursor over the desired area in the image. You will see a magnified view of that location in the **DigitalColor Meter** window. You will also see values for **R** (red), **G** (green) and **B** (blue). See Figure 8.34.

Figure 8.34

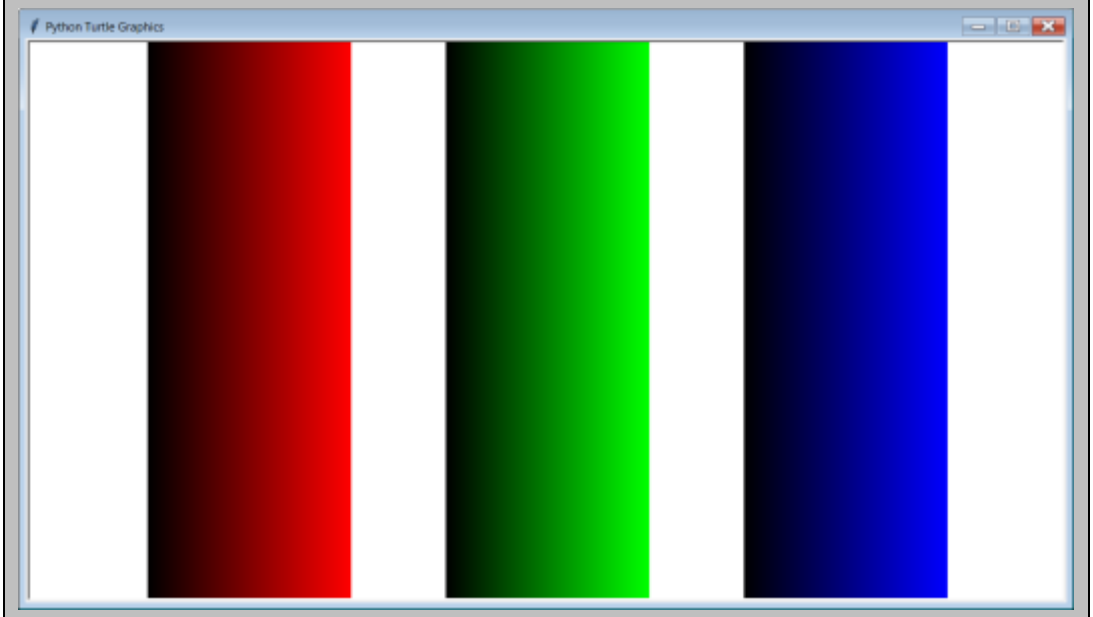


Program **CustomColors03.py**, in Figure 8.35, does something a bit different. It uses three **for** loop structures to iterate through every possible **red**, **green** and **blue** shade that can be created. Each color starts with black and then gradually changes to the most intense shade **red** or **green** or **blue**.

Figure 8.35

```
# CustomColors03.py
# This program uses the <setColor> procedure
# to show 256 shades of red, green and blue.
# This creates a "shading" effect which
# results in an illusion of depth.


from Graphics import *

beginGrfx(1300,700)

x = 150;
for red in range(256):
   setColor(red,0,0)
   drawLine(x,0,x,700)
   x += 1

x = 525;
for green in range(256):
   setColor(0,green,0)
   drawLine(x,0,x,700)
   x += 1

```

```
24 x = 900;
25 for blue in range(256):
26     setColor(0,0,blue)
27     drawLine(x,0,x,700)
28     x += 1
29
30 endGrfx()
```



If you look closely at this output, it almost looks like 3 cylindrical pillars. The brighter colors on one side and the darker colors on the other create a *shading* effect similar to what many artists use. Shading – if done properly – can give a 2-D image a 3-D appearance.

8.8 Creating Random Numbers

Generating random values is a surprisingly big deal in many ways. Billions of dollars are spent on advertising, which is largely based on the surveys of a population sample picked at random. Politicians are continuously "polling" their constituents, which are based on a proper “random selection” of potential voters. Video games have become big business and no video game can survive if the game is exactly the same every time you play it. So how do we make something random? Python has a built-in **random** library with a **randint** (short for “Random Integer”) function.

Program **RandomNumbers01.py**, in Figure 8.36, displays 5 random numbers between **1** and **100**. Do not be surprised if your output does not match mine. The program displays *random* numbers after all. Every time the program is executed, the output will be different.

Figure 8.36

```
 1 # RandomNumbers01.py
 2 # This program uses the <randint> function from
 3 # the <random> library 5 times to create 5 random
 4 # integers between 1 and 100.
 5
 6 # NOTE: When you execute the program a second time,
 7 #       you will get a different set of numbers
 8 #       because they are "random".
 9
10
11 # Required to have access to the <randint> command
12 from random import randint
13
14 print()
15 print(randint(1,100))
16 print(randint(1,100))
17 print(randint(1,100))
18 print(randint(1,100))
19 print(randint(1,100))
20
```

First Output:

```
----jGRASP exec: python RandomNumbers01.py

22
53
46
99
3

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

Second Output:

```
----jGRASP exec: python RandomNumbers01.py

58
42
30
28
42

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

Program **RandomNumbers02.py**, in Figure 8.37, displays 5 random numbers – like the previous program – however; in this program the user enters the **min**imum number and the **max**imum number. The program is also more efficient because the 5 **print** statements have been reduced to 1 **print** statement in a **for** loop.

Figure 8.37

```
1 # RandomNumbers02.py
2 # This program is more efficient and flexible than the
3 # previous program.  It is more efficient because the
4 # 5 <print> commands are now in a <for> loop.
5 # It is more flexible because the user can specify
6 # the range of random numbers.
7
```

```
 8
 9 from random import randint
10
11 print()
12 min = eval(input("Enter the smallest number.  -->  "))
13 max = eval(input("Enter the largest number.   -->  "))
14 print()
15
16 for k in range(5):
17   print(randint(min,max))
18
```

First Output:

```
   ----jGRASP exec: python RandomNumbers02.py

▶▶ Enter the smallest number.  -->  10
▶▶ Enter the largest number.   -->  99

   89
   45
   34
   40
   10

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

Second Output:

```
   ----jGRASP exec: python RandomNumbers02.py

▶▶ Enter the smallest number.  -->  1000
▶▶ Enter the largest number.   -->  9999

   6388
   1993
   8690
   3726
   2333

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

Simulations

It was mentioned earlier that random numbers are used in simulations. Program **RandomNumbers03.java**, in Figure 8.38 tries to *simulate* rolling dice 1,000,000 times. After each roll, the program checks if a **2**, **7** or **11** is rolled and keeps track of the statistics.

Figure 8.38

```
# RandomNumbers03.py
# This program INCORRECTLY simulates
# rolling dice 1,000,000 times.

from random import randint

sevens = 0
elevens = 0
snakeEyes = 0
doubles = 0

for roll in range(1000000):
    dice = randint(2,12)
    if dice == 2:
        snakeEyes += 1
    if dice == 7:
        sevens += 1
    if dice == 11:
        elevens += 1

print()
print("# of Sevens:     ",sevens)
print("# of Elevens:    ",elevens)
print("# of Snake Eyes:",snakeEyes)
print("# of Doubles:    ",doubles)

```

```
----jGRASP exec: python RandomNumbers03.py

# of Sevens:      91467
# of Elevens:     90853
# of Snake Eyes:  90756
# of Doubles:     0

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

There are a couple problems with this program. First, because of the way it was written, there is no way we can check to see if *doubles* are rolled. Second, if we look at the other statistics in the output, it gives a strong impression that the chances of rolling a 7, 11, or 2 (Snake Eyes) are the same. In reality, they are not. Look at the chart in Figure 8.39 for clarification.

Figure 8.39

Dice	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

The chart in Figure 8.39 shows all of the possible outcomes when rolling two dice. It shows there is only one way to roll a **2** (Snake Eyes). It also shows there are two ways to roll an **11** and six ways to roll a **7**. This means that the probability of rolling an **11** is double that of rolling a **2**. The probability of rolling a **7** is six times than that of rolling a **2**.

Program **RandomNumbers04.java**, in Figure 8.40 properly simulates rolling dice 1,000,000 times by handling each die (singular form of dice) separately. Not only will properly simulate the number of **2**s, **7**s, and **11**s; it will also give us a way to check for doubles.

Figure 8.40

```
 1 # RandomNumbers04.py
 2 # This program PROPERLY simulates rolling dice 1,000,000 times.
 3
 4 from random import randint
 5
 6 sevens = 0
 7 elevens = 0
 8 snakeEyes = 0
 9 doubles = 0
10
11 for roll in range(1000000):
12    die1 = randint(1,6)
13    die2 = randint(1,6)
14    diceTotal = die1 + die2
15    #
16    if diceTotal == 2:
17       snakeEyes += 1
18    if diceTotal == 7:
19       sevens += 1
20    if diceTotal == 11:
21       elevens += 1
22    if die1 == die2:
23       doubles += 1
24
25 print()
26 print("# of Sevens:    ",sevens)
27 print("# of Elevens:   ",elevens)
28 print("# of Snake Eyes:",snakeEyes)
29 print("# of Doubles:   ",doubles)
```

```
  ----jGRASP exec: python RandomNumbers04.py

 # of Sevens:     166191
 # of Elevens:    55302
 # of Snake Eyes: 27693
 # of Doubles:    166532

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

These results make more sense. The number of **11**s is about twice the numbers of *Snakes Eyes*. The number of **7**s is about 6 times the number of *Snake Eyes*. Also notice the number of **7**s is about the same as the number of *doubles*. This should also make sense because there are 6 ways to roll a **7**, and 6 ways to roll a *double*.

8.9 Using Random Numbers with Graphics

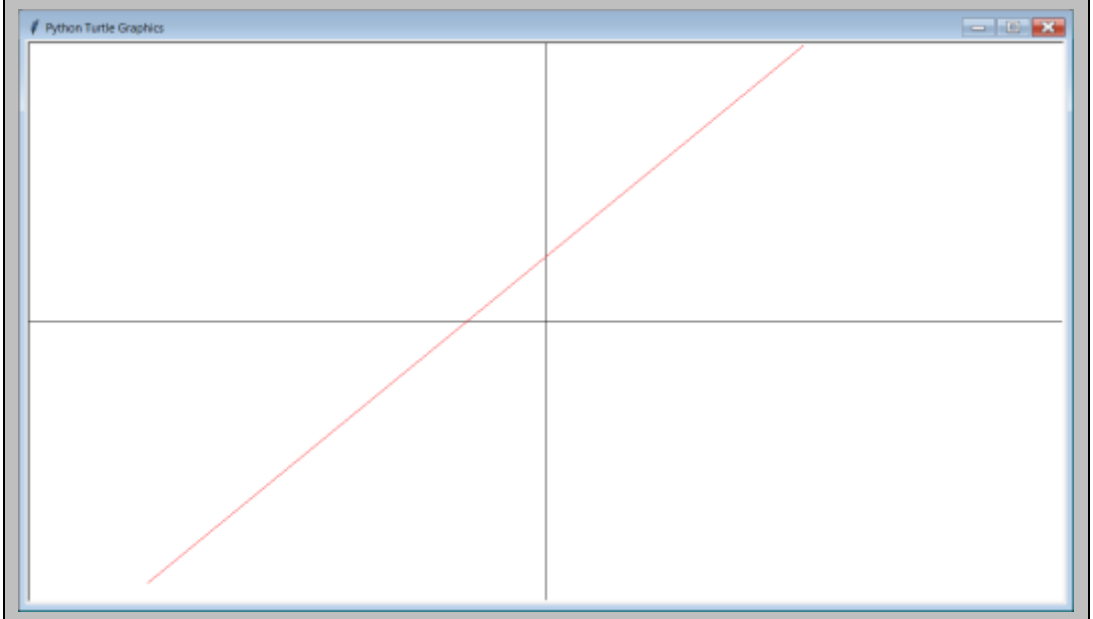
There are two programming concepts that you have learned that combine very nicely. These would be *Random Numbers* and *Graphics*. When you combine these two, you can create some interesting programs.

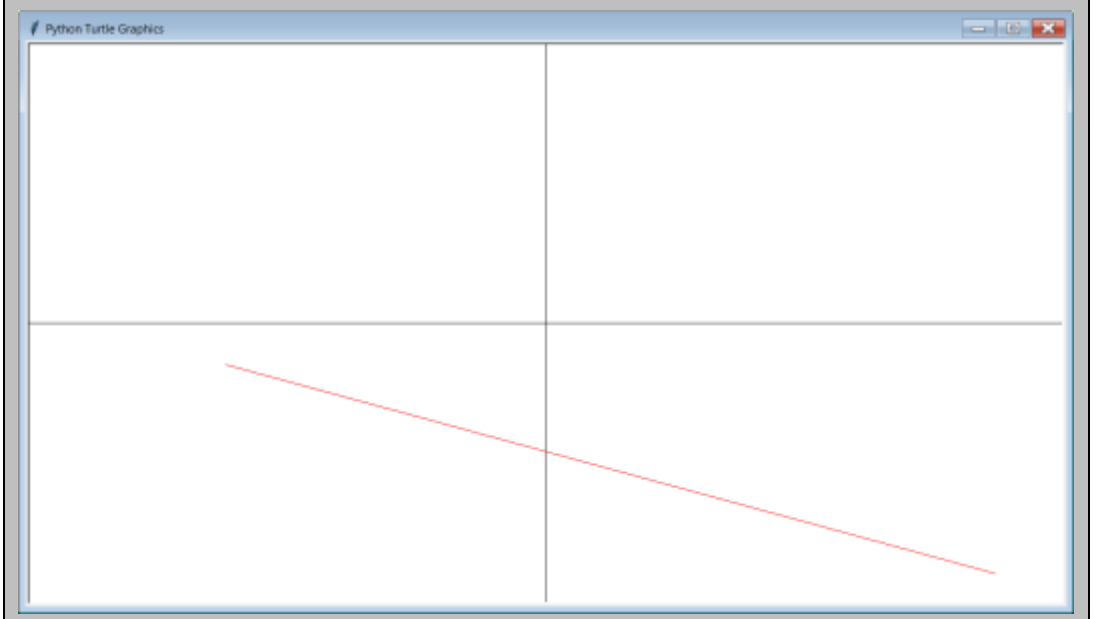
Program **RandomGraphics01.py**, in Figure 8.41, displays 3 lines. The first 2 are black and have fixed number arguments. These 2 lines will always be the same. The third line is red and has 4 integer variable arguments – and each of those integer variables stores a *random* integer value. This line will be different with each execution. Note that the range of the random integers is intentionally selected to stay within the coordinate range of the applet window.

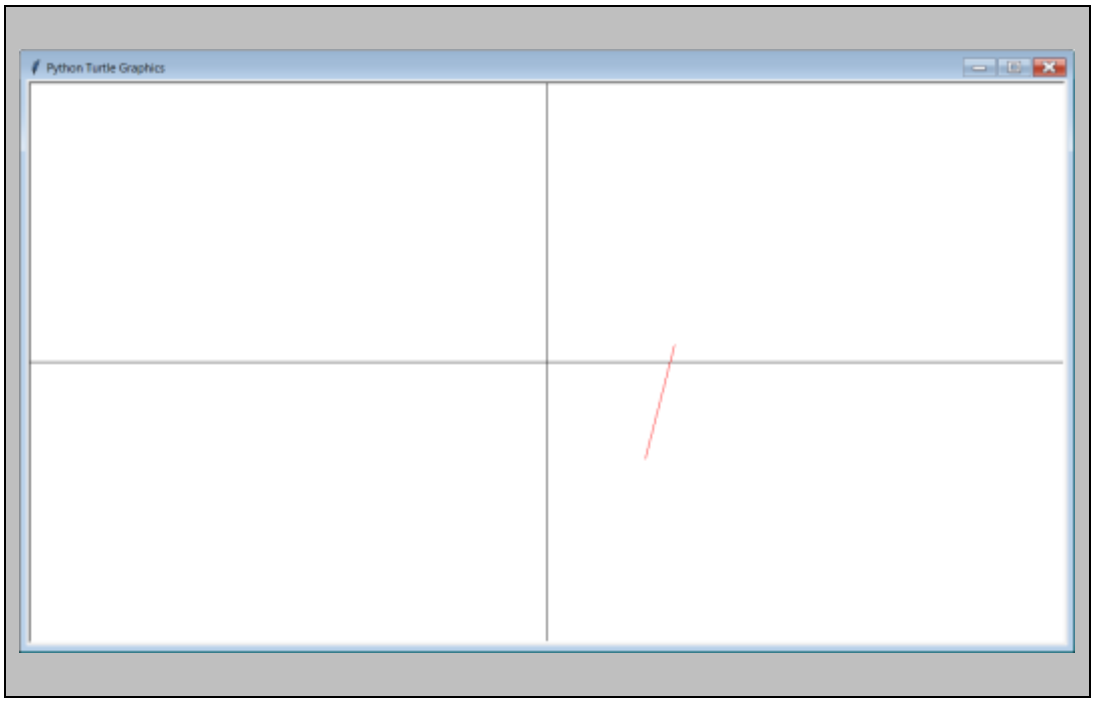
Figure 8.41

```
 1 # RandomGraphics01.py
 2 # This program first displays two black lines in a
 3 # fixed location.  This is followed by generating
 4 # four random values, which are used to draw a third,
 5 # red line in a random location.
 6
 7
 8 from Graphics import *
 9 from random import randint
10
11 beginGrfx(1300,700)
12
13 drawLine(650,0,650,700)
14 drawLine(0,350,1300,350)
15
```

```
16 setColor("red")
17 x1 = randint(0,1300)
18 y1 = randint(0,700)
19 x2 = randint(0,1300)
20 y2 = randint(0,700)
21 drawLine(x1,y1,x2,y2)
22
23 endGrfx()
```





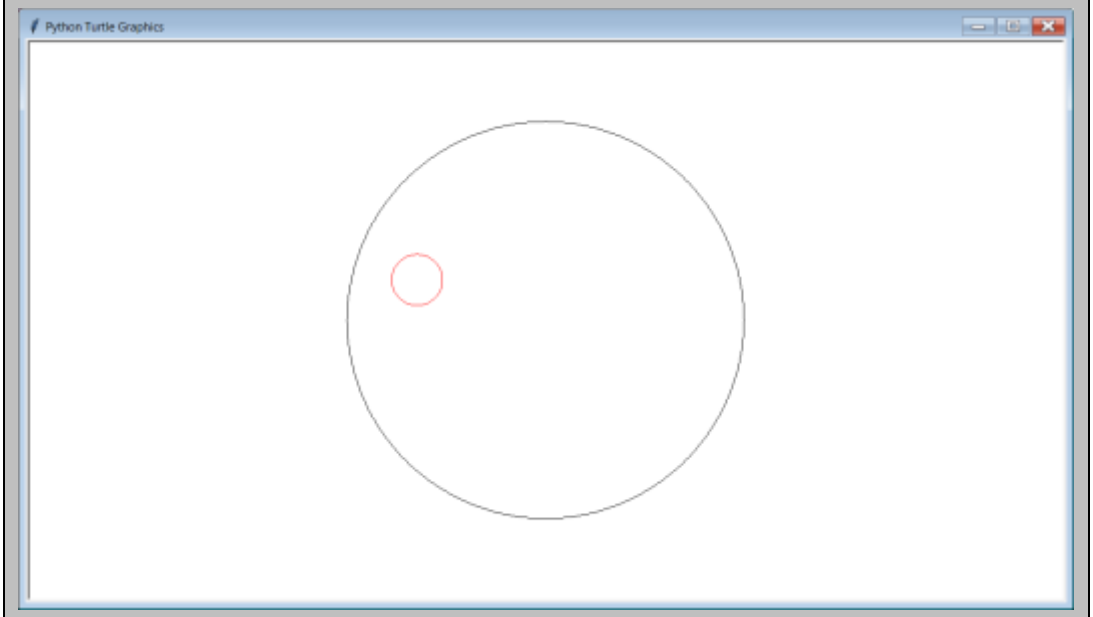


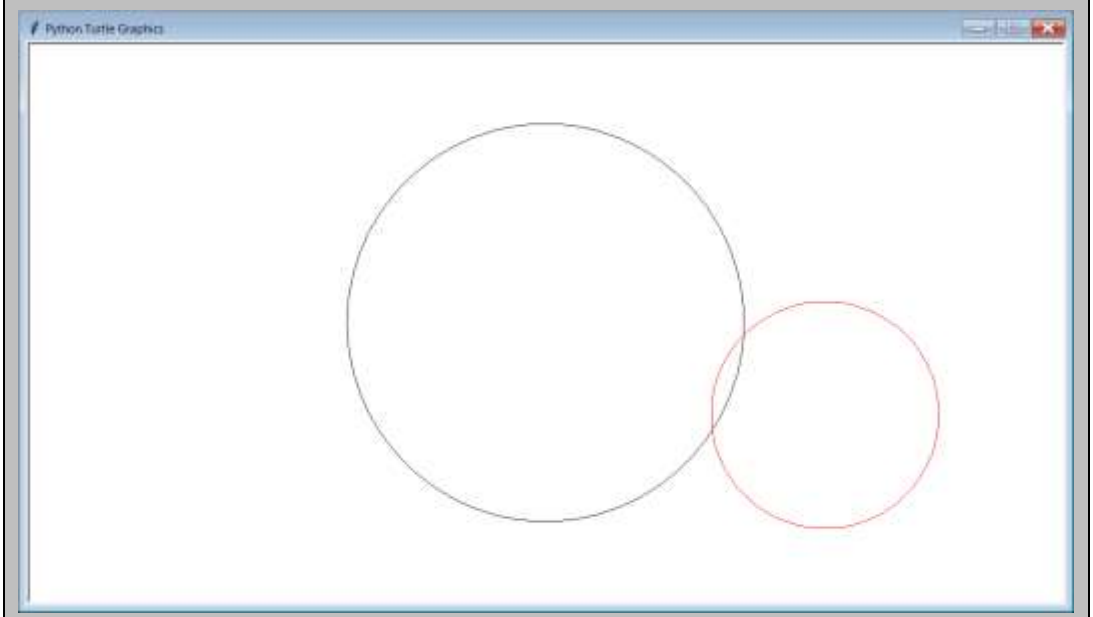
The same concept is shown in the next program. Program **RandomGraphics02.py**, in Figure 8.42, displays 2 circles. The first is black and has a fixed center coordinate and a fixed radius. This circle will always be the same. The second circle is red and uses three random integer values which determine the center (x,y) coordinate and the radius. Not only will this circle have a random location on the screen, it will also have a random size.

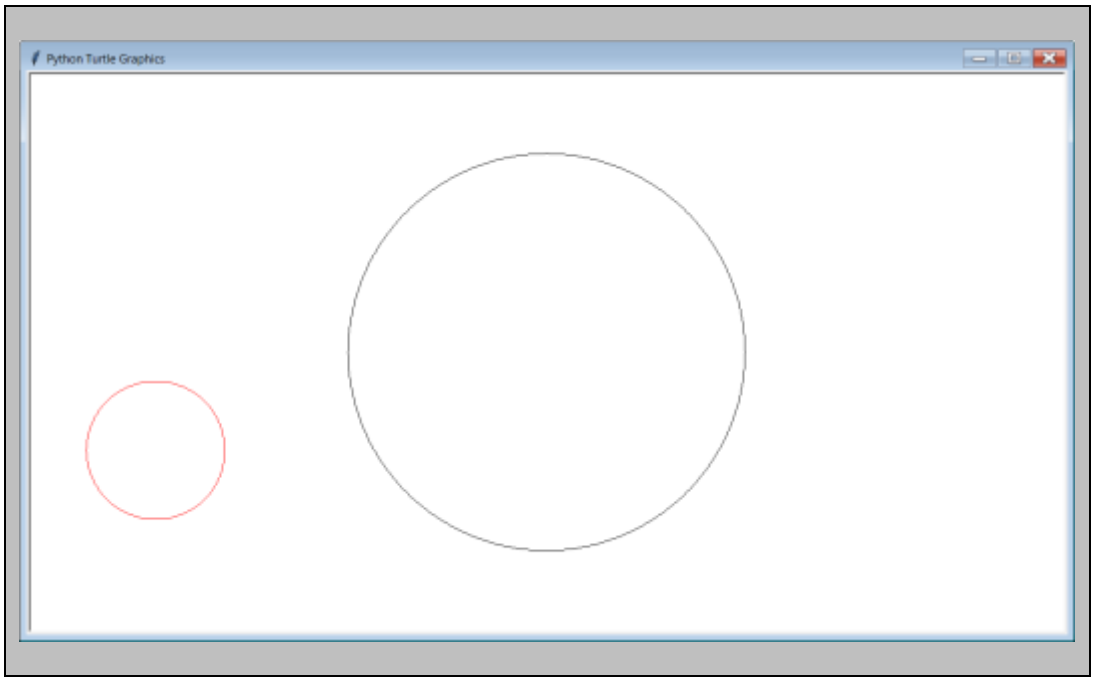
Figure 8.42

```
 1 # RandomGraphics02.py
 2 # This program displays a circle in a fixed location
 3 # with a fixed size.  3 random values are generated.
 4 # Two are used for the center location of the circle
 5 # and a third is used for its radius.
 6
 7
 8 from Graphics import *
 9 from random import randint
10
11 beginGrfx(1300,700)
12
```

```
13 drawCircle(650,350,250)
14
15 setColor("red")
16 x = randint(150,1150)
17 y = randint(150,550)
18 r = randint(10,150)
19 drawCircle(x,y,r)
20
21 endGrfx()
```







Now we will add loops to the mix. Program **RandomGraphics03.py**, in Figure 8.43, generates **1000** random lines. All of these lines are all **black**.

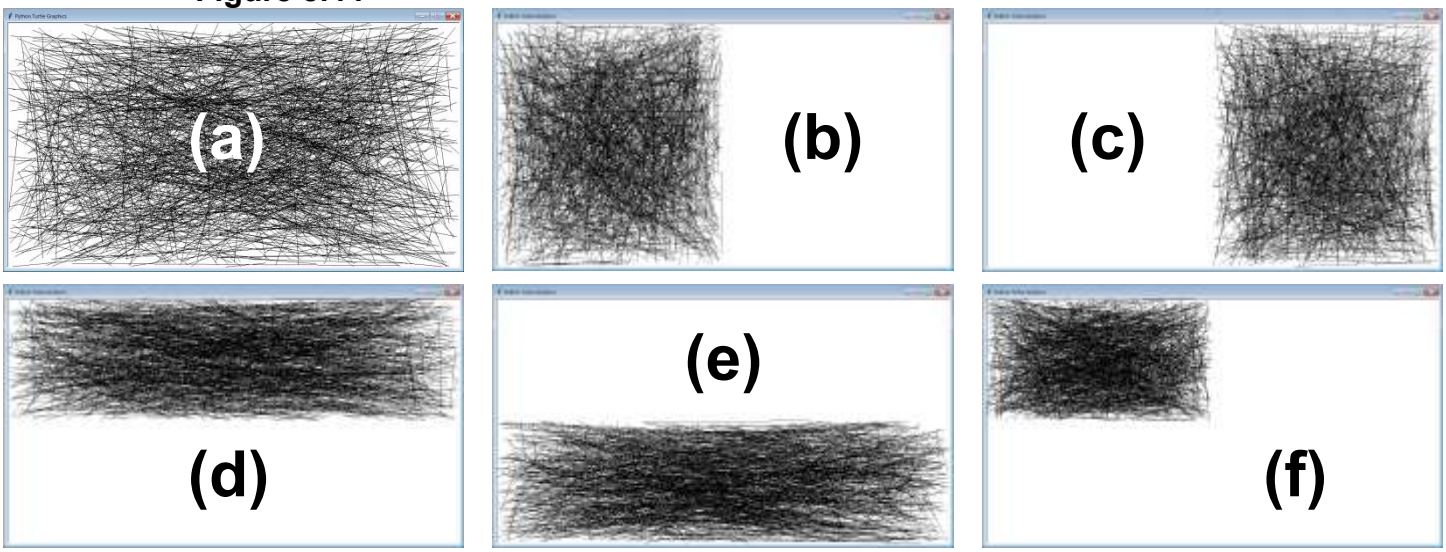
Figure 8.43

```
# RandomGraphics03.py
# This program displays 1000 random lines.


from Graphics import *
from random import randint

beginGrfx(1300,700)

for k in range(1000):
    x1 = randint(0,1300)
    y1 = randint(0,700)
    x2 = randint(0,1300)
    y2 = randint(0,700)
    drawLine(x1,y1,x2,y2)

endGrfx()
```

Changing Random Number Ranges to Affect the Graphics Output

If you alter the ranges of the random numbers in program **RandomGraphics03.py**, you will change the program's output. Consider the 6 different outputs in Figure 8.44. Figure 8.42**a** shows the current output of program **RandomGraphics03.py**. Can you figure out how to alter the random number ranges to produce the outputs in Figure 8.45**b** through 8.45**f**? Your teacher will probably go over this in class. This will be an essential skill in completing Lab 8C.

Figure 8.44



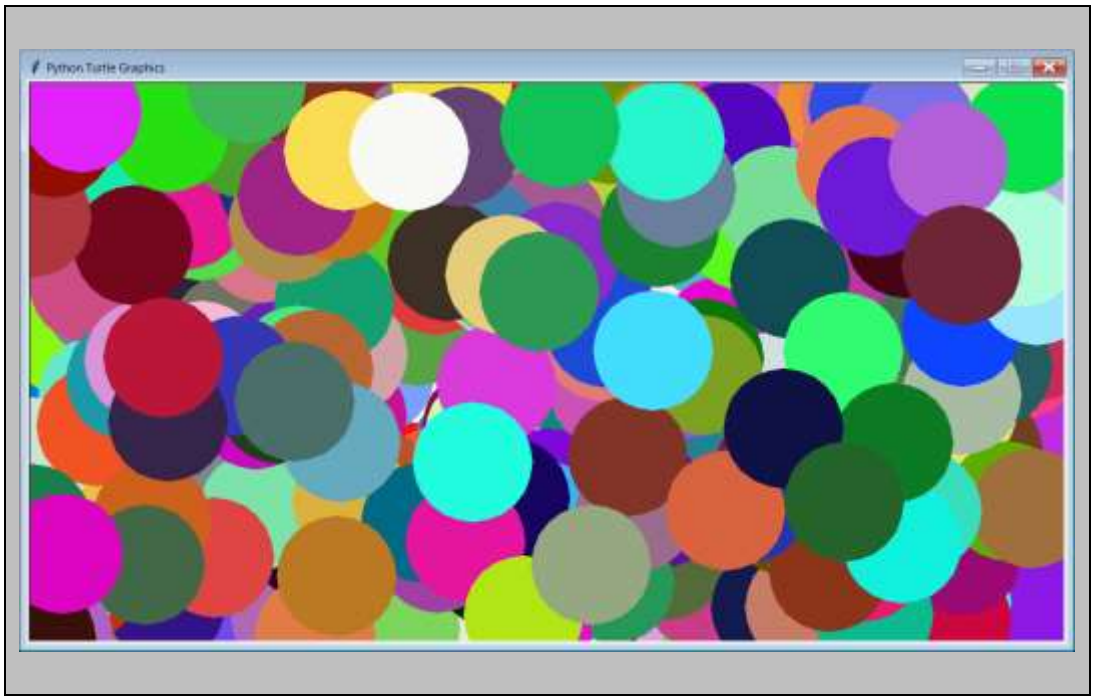
Random Colors

The idea of a *random color* may seem a bit strange. Random numbers, OK; but random colors… how is that possible? Remember that colors can be created by using 3 integer values between 0 and 255. Well, what if these 3 values are *random*. Program **RandomGraphics04.py**, in Figure 8.45, generates random **red**, **green** and **blue** integer values, each between **0** and **255**. When these random amounts of colors are *mixed* the result is a *random color*. These random colors are used to draw **200** random solid circles. Every circle has a radius of **75**.

There is something you need to realize before you execute the remaining programs in this chapter. They may "take a while" to execute. In fact, when you execute them, you may think the computer is jammed. Just be patient. Wait about 20 seconds, and eventually the output will display.

Figure 8.45

```
 1 # RandomGraphics04.py
 2 # This program displays 500 randomly colored
 3 # solid circles with a fixed radius of 75.
 4
 5
 6 from Graphics import *
 7 from random import randint
 8
 9 beginGrfx(1300,700)
10
11 for k in range(500):
12     x = randint(0,1300)
13     y = randint(0,700)
14     red = randint(0,255)
15     green = randint(0,255)
16     blue = randint(0,255)
17     setColor(red,green,blue)
18     fillCircle(x,y,75)
19
20 endGrfx()
21
```

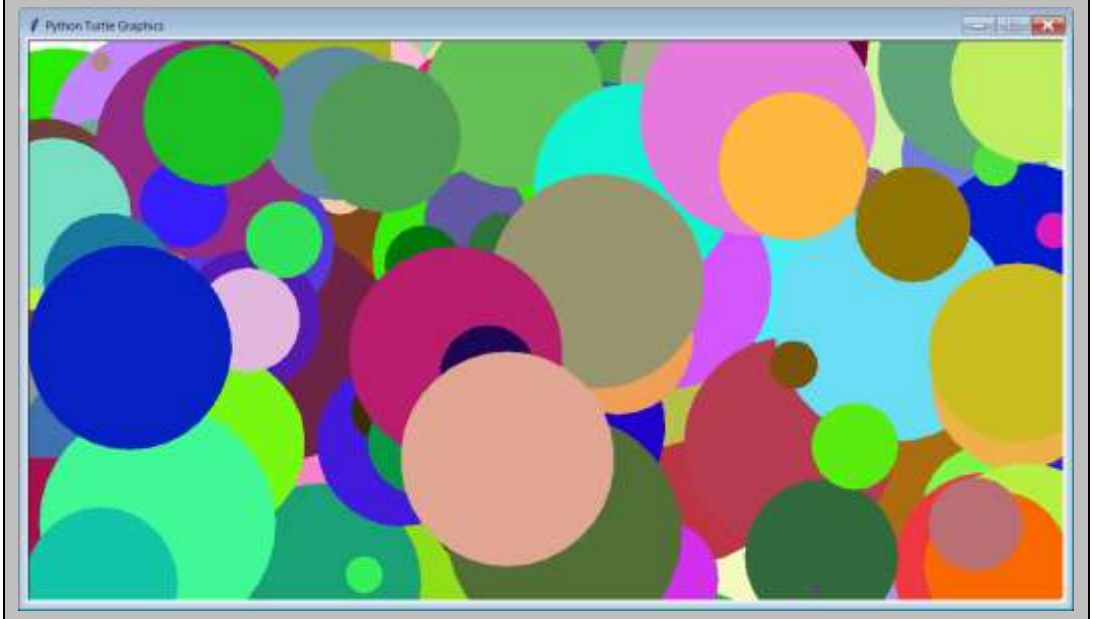


Program **RandomGraphics05.py**, in Figure 8.46, is almost the same as the previous program. The one difference is now the **radius** is random as well. Note how that affects the output.

Figure 8.46

```
# RandomGraphics05.py
# This program displays 500 randomly colored
# solid circles with random radii.


from Graphics import *
from random import randint

beginGrfx(1300,700)

for k in range(500):
    x = randint(0,1300)
    y = randint(0,700)
```

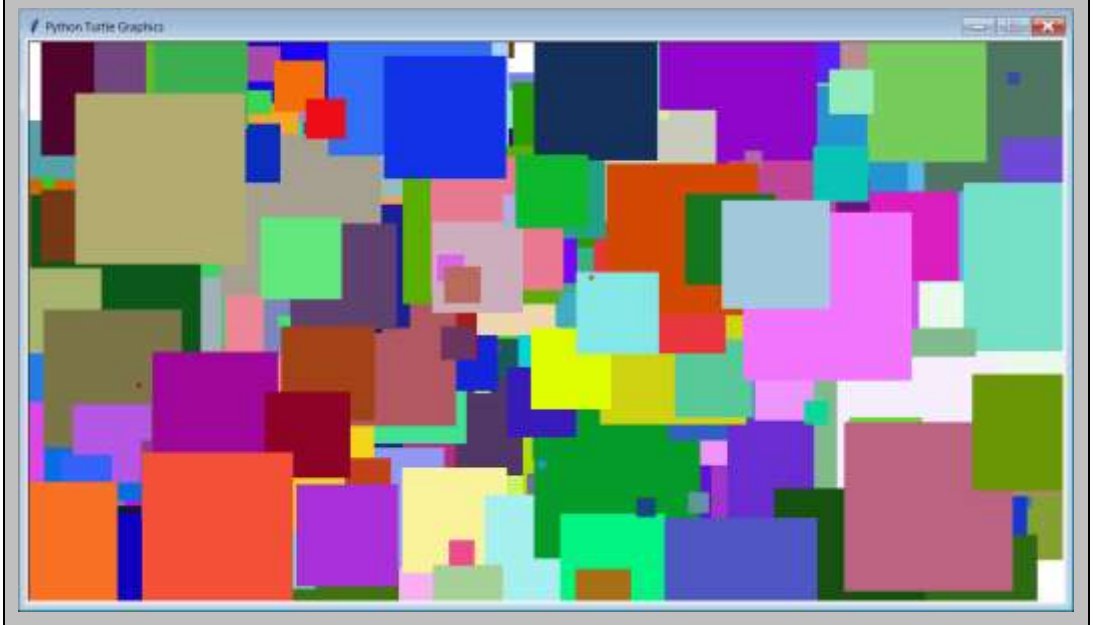
```
14      red = randint(0,255)
15      green = randint(0,255)
16      blue = randint(0,255)
17      radius = randint(1,150)
18      setColor(red,green,blue)
19      fillCircle(x,y,radius)
20
21  endGrfx()
22
```



Program **RandomGraphics06.py**, in Figure 8.47, changes from circles to regular polygons. It displays **500** regular polygons – which all turn out to be *squares*. Several things are random in this program. The values of **x** and **y** are random, which means that every regular polygon has a random location. The **radius** is random, so every regular polygon has a random size. Even the color is random, although in this program random colors are created with a single **setRandomColor** command. The one thing that is not random is the number of **sides**. It is always **4**. This is why every random polygon is a *square*.

Figure 8.47

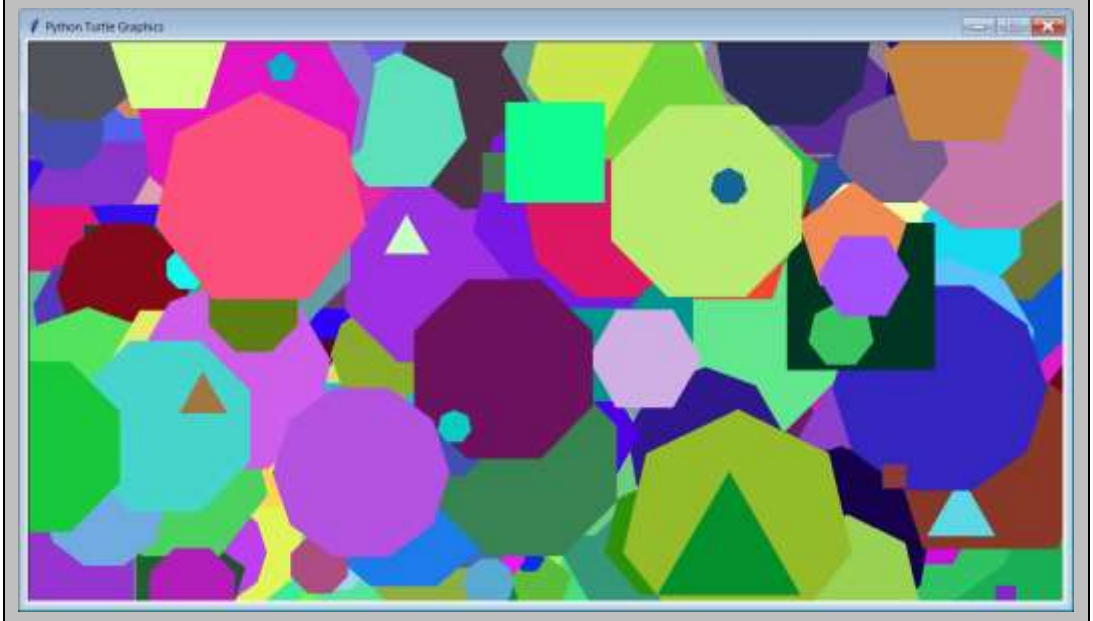
```
# RandomGraphics06.py
# This program displays 500 randomly colored squares.
# Creating the random colors is also simplified
# with the <setRandomColor> procedure.


from Graphics import *
from random import randint

beginGrfx(1300,700)

for k in range(500):
    x = randint(0,1300)
    y = randint(0,700)
    radius = randint(1,150)
    sides = 4
    setRandomColor()
    fillRegularPolygon(x,y,radius,sides)

endGrfx()

```



Now look at program **RandomGraphics07.py**, in Figure 8.48. The one difference between this program and the previous program is now the number of **sides** is random. This means we will not only see *squares*, we will also see *equilateral triangles*, *regular pentagons*, *regular hexagons*, all the way up to *regular decagons*.

Figure 8.48

```
# RandomGraphics07.py
# This program displays 500 randomly colored
# polygons with a random number of sides.


from Graphics import *
from random import randint

beginGrfx(1300,700)

for k in range(500):
    x = randint(0,1300)
    y = randint(0,700)
    radius = randint(1,150)
    sides = randint(3,10)
    setRandomColor()
    fillRegularPolygon(x,y,radius,sides)

endGrfx()
```



Program **RandomGraphics08.py**, in Figure 8.48, is the final program in this chapter. The program will display **500** *bursts*. In addition to a random location, a random radius, and a random color for each snowflake looking *burst*, the *burst* design will also have a random number of lines, and those lines will have a random **width**.

Figure 8.48

```
# RandomGraphics08.py
# This program demonstrates that even the
# width of the lines can be random.


from Graphics import *
from random import randint

beginGrfx(1300,700)

for k in range(500):
    x = randint(0,1300)
    y = randint(0,700)
    radius = randint(1,150)
    numLines = randint(3,10)
    setRandomColor()
    w = randint(1,30)
    width(w)
    drawBurst(x,y,radius,numLines)

endGrfx()
```