Chapter VI

More Python Libraries

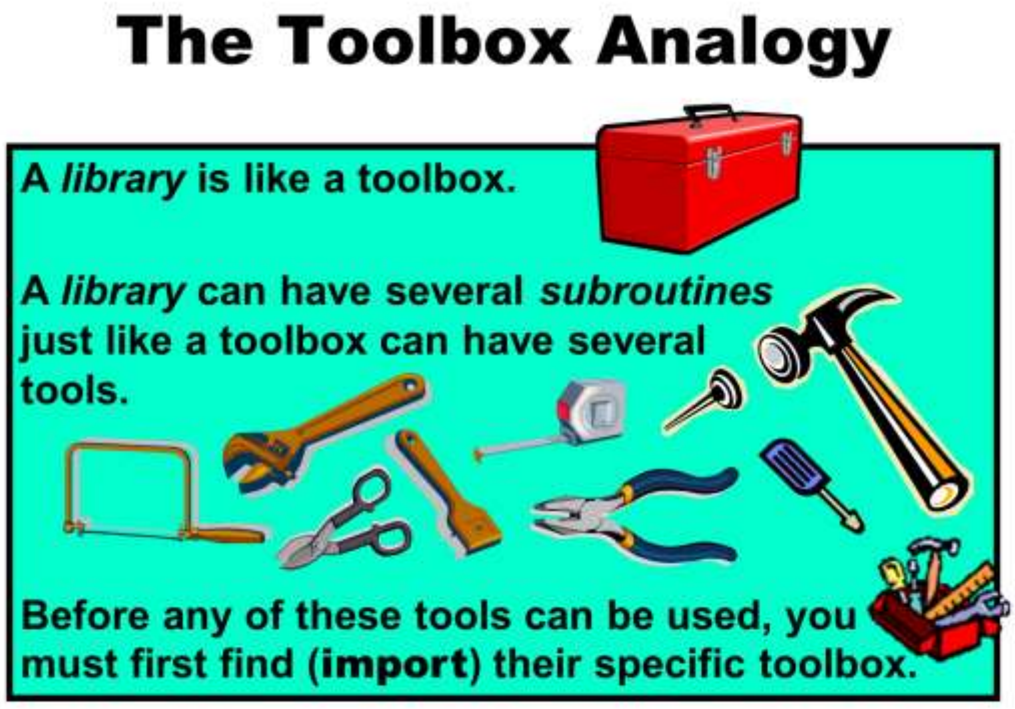
Chapter VI Topics

6.1 Introduction

In Chapter V, we looked at *Turtle Graphics*. While this was an introduction to the concept of *graphics* it was also an introduction to the practice of *importing libraries*. We imported commands from both the **turtle** library and the **time** library. Python has many, many more libraries. We will look at a couple more libraries in this chapter.

6.2 Library Components

Before we start looking at program examples that import other libraries, we need to better understand exactly they contain. In general, libraries contain several *subroutines*. A subroutine is a series of program commands that performs a specific task. The analogy below may help to better explain the difference between libraries and subroutines:


The Toolbox Analogy
A *library* is like a toolbox.
A *library* can have several *subroutines* just like a toolbox can have several tools.
Before any of these tools can be used, you must first find (**import**) their specific toolbox.

Functions and Procedures

There are different types of subroutine that can be used in Python. In this first year class, we focus on the 2 most commonly used. These are *functions* and *procedures*. While both of these describe "a series of program commands that performs a specific task," there is one very important difference. *Functions* return a value and *procedures* do not.

It may help is you think of a *mathematical function* which has both input and output. In this example the math function is **f(x) = 2x + 3**.

Figure 6.1

$$\text{input }(x) \rightarrow \boxed{2x + 3} \rightarrow \text{output}$$

So if **x** equals **10**, we get:

$$10 \rightarrow \boxed{2(10) + 3} \rightarrow 23$$

which shows us that for this *function*, **f(10) = 23**.

So *functions* perform a series of program commands, typically some type of mathematical calculation, and then essentially *return* the final answer. *Procedures* also perform a series of program commands, but there is no "final answer" to return.

For right now, you are going to use functions and procedures that have been created for you. When we get to chapter X, you will be creating your own.

Subroutines, Functions and Procedures

A *subroutine* is a series of programming commands that performs a specific task.

A *function* is a subroutine that returns a value.

A *procedure* is a subroutine that does not return a value.

6.3 math Library Functions

The **math** library is full of several mathematical functions. Program **mathLibrary01.py**, in Figure 6.2, starts with the **sqrt** function. “**sqrt**” is an abbreviation for *square root*. On line 12, we import all of the **math** library functions. The **sqrt** function is called twice. The first time is on line 16 with an integer *argument*. The second time is on line 18 with a real number *argument*. The **sqrt** function actually works with both integers and real numbers. The term *argument* refers to the information that a function or procedure requires. This is like the *input* shown in Figure 6.1.

Figure 6.2

```
 1 # mathLibrary01.py
 2 # This program demonstrates the <sqrt> function
 3 # of the <math> library, which returns the
 4 # principal square root of the argument.
 5
 6 # NOTE: Most of the functions in the <math> library
 7 #       work with both integers and real numbers.
 8
 9
10 # Required to have access to most
11 # of the <math> library functions.
12 from math import *
13
14 print()
15
16 print("The square root of 625 is",sqrt(625))
17
18 print("The square root of 6.25 is",sqrt(6.25))
19
```

```
----jGRASP exec: python mathLibrary01.py

The square root of 625 is 25.0
The square root of 6.25 is 2.5

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

Suppose you type only **sqrt()**, with no argument. Can Python digest such a statement? Can you? What do you say when your math teacher walks up to you and says: *"Give me the square root!"* You will probably be a little perplexed and respond *"Of what?"* because the computation of a mathematical problem requires some type of information. In Python, this information is *passed* to the function or procedure, which can then provide the requested response.

Program **mathLibrary02.py**, in Figure 6.3, demonstrates that arguments can be passed in different formats. It is possible to pass a constant, a variable, an expression or even the value returned by another function.

Figure 6.3

```python
# mathLibrary02.py
# This program shows several different arguments
# that can be used with the <sqrt> function.
# Note how one function call can be the argument
# of another function call.


from math import *

n1 = sqrt(1024)        # constant argument
n2 = sqrt(n1)          # variable argument
n3 = sqrt(n1 + n2)     # expression argument
n4 = sqrt(sqrt(256))   # function argument

print()
print("n1:",n1)
print("n2:",n2)
print("n3:",n3)
print("n4:",n4)
```

```
 ----jGRASP exec: python mathLibrary02.py

n1: 32.0
n2: 5.656854249492381
n3: 6.136518088418903
n4: 4.0

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

Issues with Arguments

You have seen that **sqrt** can find the square root of both integers and real numbers. What would happen if the **sqrt** argument was not the right kind of information? What if you tried to find the square root of a negative number? What is you tried to find the square root of a string value. These are the issues that we will investigate in the next couple programs.

Program **mathLibrary03.py**, shown in Figure 6.4, demonstrates what happens when the argument is not the correct data type. The program basically tries to find the square root of the word **"Fish"** which simply does not work. The syntax error indicates that the type must be a real number, not a string.

Figure 6.4

```
# mathLibrary03.py
# This program demonstrates what happens when a
# function is called with the wrong data type.


from math import *

n = sqrt("Fish")  # invalid argument

print()
print("The square root of 'Fish' is:",n)

```

```
 ----jGRASP exec: python mathLibrary03.py
Traceback (most recent call last):
  File "mathLibrary03.py", line 9, in <module>
    n = sqrt("Fish")  # invalid argument
TypeError: must be real number, not str

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

When dealing with arguments, sometimes having the correct *datatype* is not good enough. Program **mathLibrary04.py**, shown in Figure 6.4, calls the **sqrt** function again. This time there is a *number* as the argument. The problem is it is negative. If you have paid attention in your math classes you should know that you cannot take the square root of a negative number.

NOTE: For those of you in higher-level math classes. We will not be working with *Imaginary* or *Complex Numbers* in this computer science class.

Unlike the previous program, this program does execute. There are no syntax errors. The interpreter sees that the argument is a number (integer or real number) and is happy.

Figure 6.5

```python
# mathLibrary04.py
# This program demonstrates what happens when
# you take the square root of a negative number
# in Python. It causes a Run-time Error, similar
# to what happens when you divide by zero.


from math import *

print()
print("Execution Begins")

n = sqrt(-1)  # invalid argument

print()
print("The square root of -1 is ",n)

```

```
 ----jGRASP exec: python mathLibrary04.py

Execution Begins
Traceback (most recent call last):
  File "mathLibrary04.py", line 13, in <module>
    n = sqrt(-1)  # invalid argument
ValueError: math domain error

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

When you look at the output, you can see that the program did at least partially execute because the words "**Execution Begins**" are displayed. This output, and the blank line before it, were generated by the **print** commands in lines 10 and 11. However, on line 13, when the computer attempted to compute the square root of a negative number, the program crashed in a manner similar to when we attempted to divide by zero a couple chapters ago.

Subroutine Arguments

The information, which is passed to a subroutine (function or procedure) is called an *argument*.

Arguments are placed between parentheses immediately following the subroutine identifier.

Arguments can be constants, variables, expressions or they can be other function calls.

The only requirement is that the correct data type value is passed to the subroutine.

In other words, **sqrt(x)** can compute the square root of **x**, if **x** stores any non-negative number (integer or real number), but not if **x** stores a negative number or string value like **"Fish"**.

Absolute Value Function abs

Program **mathLibrary05.py**, in Figure 6.6, demonstrates the **abs** function which is used the compute the *Absolute Value* of the argument. The output of the program should confirm what you already know about absolute value; specifically, the absolute value of any non-negative number is the number itself. If the number is negative, absolute value removes the negative sign and essentially makes it positive.

This program also demonstrates how absolute value can be used to protect against taking the square root of a negative number. Suppose you have an unknown number and you want to compute its square root. The problem is the number could be negative. The solution, first compute the absolute value and then compute the square root of the result.

Figure 6.6

```python
# mathLibrary05.py
# This program demonstrates the <abs>
# function, which returns the absolute
# value of the argument.  It also shows
# how the <abs> function can be used to
# prevent the issue of a negative argument
# with the <sqrt> function.


from math import *

print()
print("The absolute value of -25 is",abs(-25))
print("The absolute value of 100 is",abs(100))
print("The absolute value of 3.7 is",abs(3.7))
print("The absolute value of -.5 is",abs(-.5))
print("The absolute value of 0.0 is",abs(0.0))
print()
print("The square root of the absolute value")
print("of -256 is",sqrt(abs(-256)))

```

```
 ----jGRASP exec: python mathLibrary05.py

The absolute value of -25 is 25
The absolute value of 100 is 100
The absolute value of 3.7 is 3.7
The absolute value of -.5 is 0.5
The absolute value of 0.0 is 0.0

The square root of the absolute value
of -256 is 16.0

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

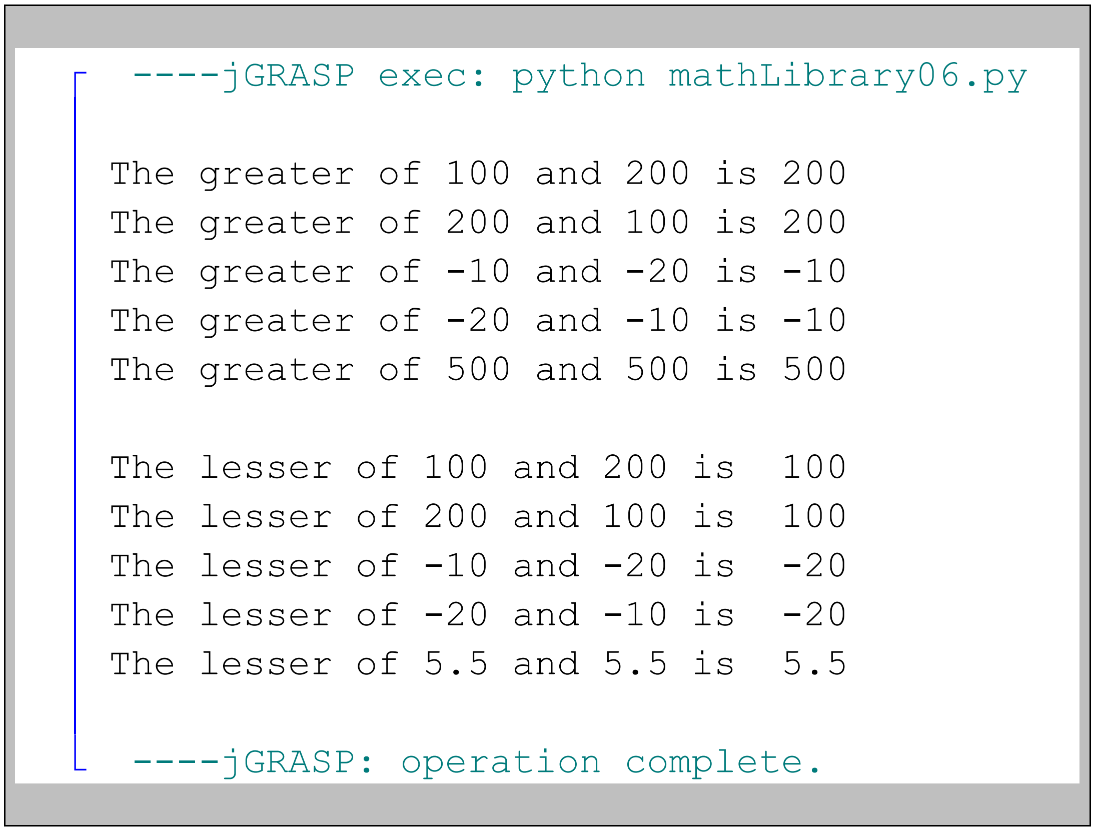
max and min Functions

The first 5 program examples can easily give the impression that functions use a single argument, or at least it appears that function of the **math** library use a single argument. In mathematics there are many examples where only a single argument is required. Functions like square root and absolute value are computed with a single value. There are also some examples where multiple arguments or arguments are used. Many area and volume computations involve multiple arguments, like the area of a rectangle, which requires **length** and **width**. Python's **math** library has a few functions which require two arguments. Two of these functions are **max** and **min**. The **max** procedure returns the larger of the two arguments and the **min** procedure returns the smaller. Both functions are demonstrated in program **mathLibrary06.py**, shown in Figure 6.7.

Keep in mind, **max** and **min** are two of the simplest functions in the **math** library; however, students often get confused when the numbers are negative. **max** returns the number that is "greater than" the other number, even if that number has a smaller "absolute value". (Think of a number line like the type you saw in elementary/middle school.) **min** return the number that is "less than" the other.

Figure 6.7

```
# mathLibrary06.py
# This program demonstrates the <max> and <min>
# functions of the <math> library.
# <max> returns the greater of the two arguments.
# <min> returns the lesser of the two arguments.


from math import *

print()
print("The greater of 100 and 200 is",max(100,200))
print("The greater of 200 and 100 is",max(200,100))
print("The greater of -10 and -20 is",max(-10,-20))
print("The greater of -20 and -10 is",max(-20,-10))
print("The greater of 500 and 500 is",max(500,500))
print()
print("The lesser of 100 and 200 is ",min(100,200))
print("The lesser of 200 and 100 is ",min(200,100))
print("The lesser of -10 and -20 is ",min(-10,-20))
print("The lesser of -20 and -10 is ",min(-20,-10))
print("The lesser of 5.5 and 5.5 is ",min(5.5,5.5))

```



```
 ----jGRASP exec: python mathLibrary06.py

The greater of 100 and 200 is 200
The greater of 200 and 100 is 200
The greater of -10 and -20 is -10
The greater of -20 and -10 is -10
The greater of 500 and 500 is 500

The lesser of 100 and 200 is  100
The lesser of 200 and 100 is  100
The lesser of -10 and -20 is  -20
The lesser of -20 and -10 is  -20
The lesser of 5.5 and 5.5 is  5.5

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

Computing Exponents with Function pow

Some programming languages are created by one single person. Pascal, for example, was created by a college professor named Niklaus Wirth. As a result, Pascal is a very "lean" language with nothing extra or redundant. Most modern languages today are too big to be created by just one person. Instead, they are created by a team of people. Sometimes, completely different people update the language years later. As a result, these languages can sometime have some features that are redundant. Python is one such language.

You have already learned that the double asterisk (`**`) operator is used for exponents in Python. Python's **math** library also has a special function for exponents called **pow**. This is short for "power" as in "taking one number to the power of another". Like **max** and **min**, the **pow** function actually has 2 arguments. The first argument is for the *base* and the second argument is for the *exponent*. With **max** and **min** the order of the arguments did not matter; however, that is usually not the case. The order of the arguments definitely matters with the **pow** function because it computes the first argument to the power of the second. This is demonstrated in program **mathLibrary07.py**, shown in Figure 6.8.

Figure 6.8

```
# mathLibrary07.py
# This program demonstrates the <pow> function
# of the <math> library which does the same
# thing as the exponent operator <**>.

# NOTE: Like <max> and <min>, <pow> uses 2
# arguments.  However, unlike <max> and <min>,
# with <pow> the order of the 2 arguments is
# VERY significant.
# The first argument is the "base".
# The second argument is the "exponent".
# <pow> returns the first argument to the
# "power" of the second argument.


from math import *

print()
print("3 to the 4th power using ** is",3 ** 4)
print("3 to the 4th power with pow is",pow(3,4))
print()
print("4 to the 3rd power using ** is",4 ** 3)
print("4 to the 3rd power with pow is",pow(4,3))

```

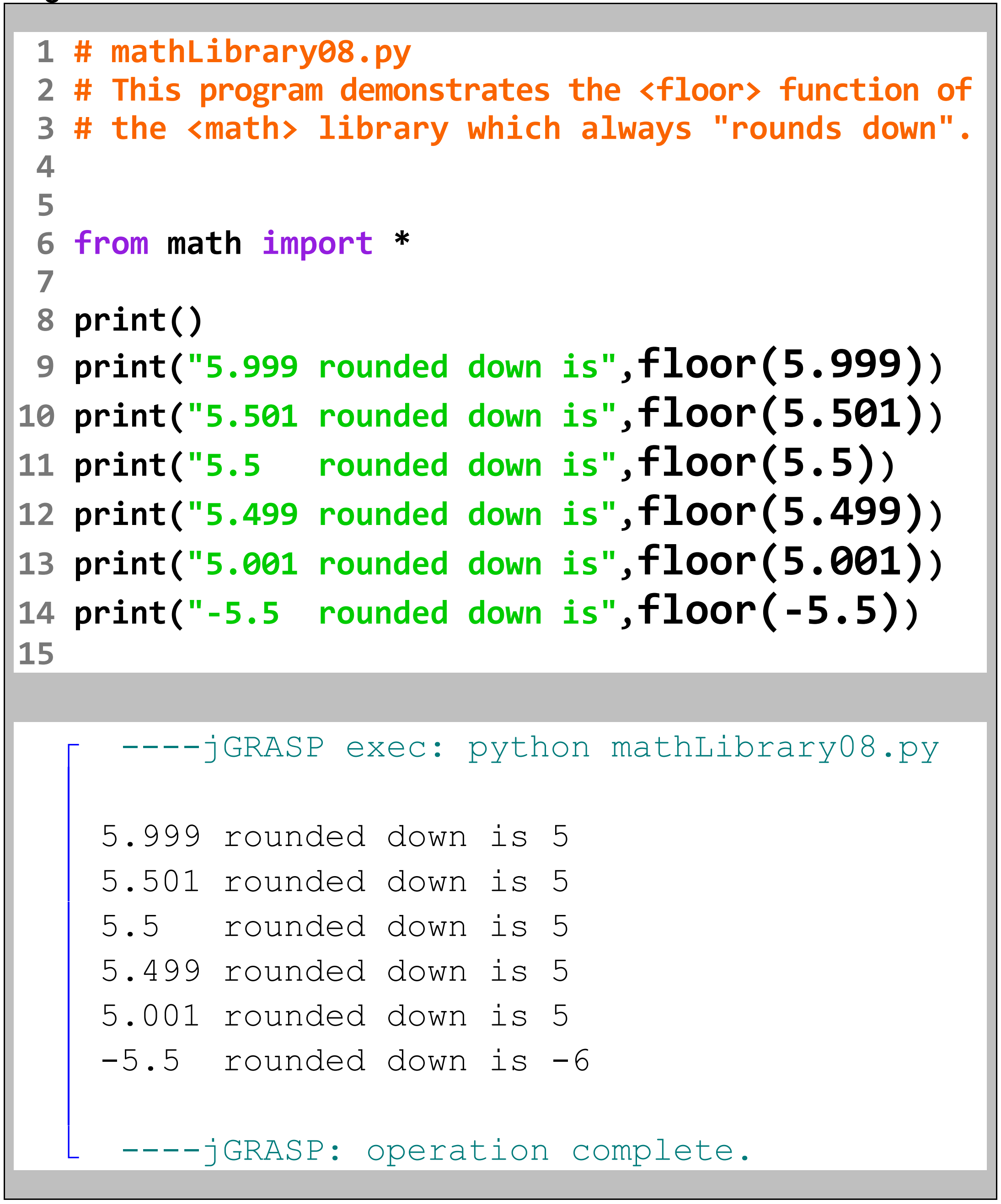
```
 ----jGRASP exec: python mathLibrary07.py

3 to the 4th power using ** is 81
3 to the 4th power with pow is 81.0

4 to the 3rd power using ** is 64
4 to the 3rd power with pow is 64.0

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

Different Types of "Rounding"

Rounding numbers is a practical function in mathematics. The **math** library actually has 4 different rounding functions. Each of these "rounds" in a different way and has a separate program example.

Program **mathLibrary08.py**, in figure 6.9, demonstrates the **floor** function. The function is called **floor** because the "floor" is below you. This is to help you remember that this function always rounds down.

Figure 6.9



```python
# mathLibrary08.py
# This program demonstrates the <floor> function of
# the <math> library which always "rounds down".


from math import *

print()
print("5.999 rounded down is",floor(5.999))
print("5.501 rounded down is",floor(5.501))
print("5.5   rounded down is",floor(5.5))
print("5.499 rounded down is",floor(5.499))
print("5.001 rounded down is",floor(5.001))
print("-5.5  rounded down is",floor(-5.5))

```

```
 ----jGRASP exec: python mathLibrary08.py

5.999 rounded down is 5
5.501 rounded down is 5
5.5   rounded down is 5
5.499 rounded down is 5
5.001 rounded down is 5
-5.5  rounded down is -6

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

Program **mathLibrary09.py**, in figure 6.10, demonstrates the **ceil** function. This is short for “ceiling”. In the same way that the “floor” is below you, the “ceiling” is always above you. This is to help you remember that **ceil** always rounds up.

NOTE: In your math class, the *Greatest Integer Function* is written with the notation **y = [x]**. This is exactly what Python’s **ceil** function computes.

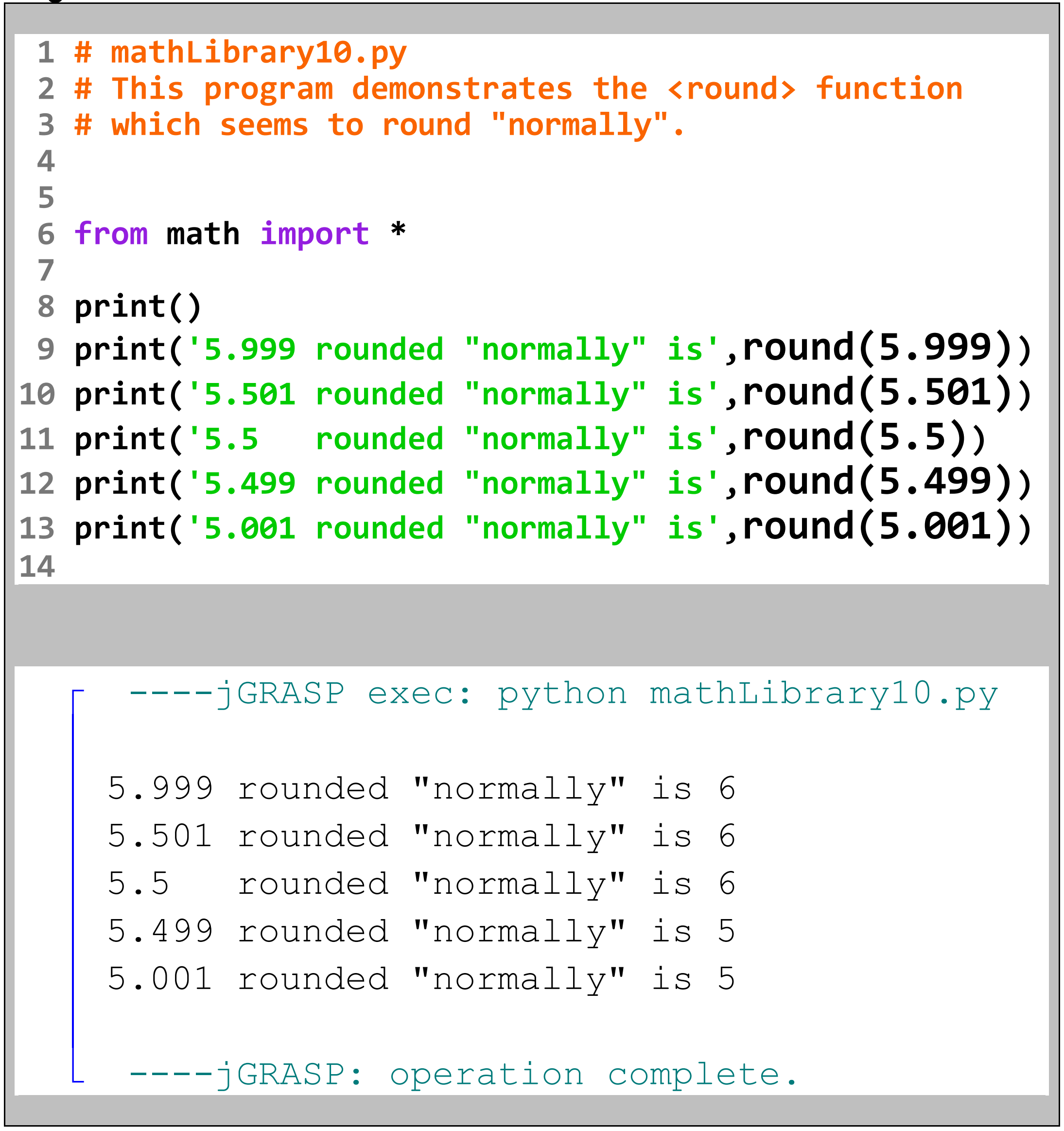
Figure 6.10

```python
# mathLibrary09.py
# This program demonstrates the <ceil> function of
# the <math> library which always "rounds up".


from math import *

print()
print("5.999 rounded up is",ceil(5.999))
print("5.501 rounded up is",ceil(5.501))
print("5.5   rounded up is",ceil(5.5))
print("5.499 rounded up is",ceil(5.499))
print("5.001 rounded up is",ceil(5.001))
print("-5.5  rounded up is",ceil(-5.5))

```

```
 ----jGRASP exec: python mathLibrary09.py

5.999 rounded up is 6
5.501 rounded up is 6
5.5   rounded up is 6
5.499 rounded up is 6
5.001 rounded up is 6
-5.5  rounded up is -5

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

You may be thinking, “What if I don’t want to *round up* or *round down*? What if I just want to *round normally*?” Program **mathLibrary10.py**, in figure 6.11, demonstrates the **round** function. When you look at the output of this program, the **round** function seems to round normally.

Figure 6.11



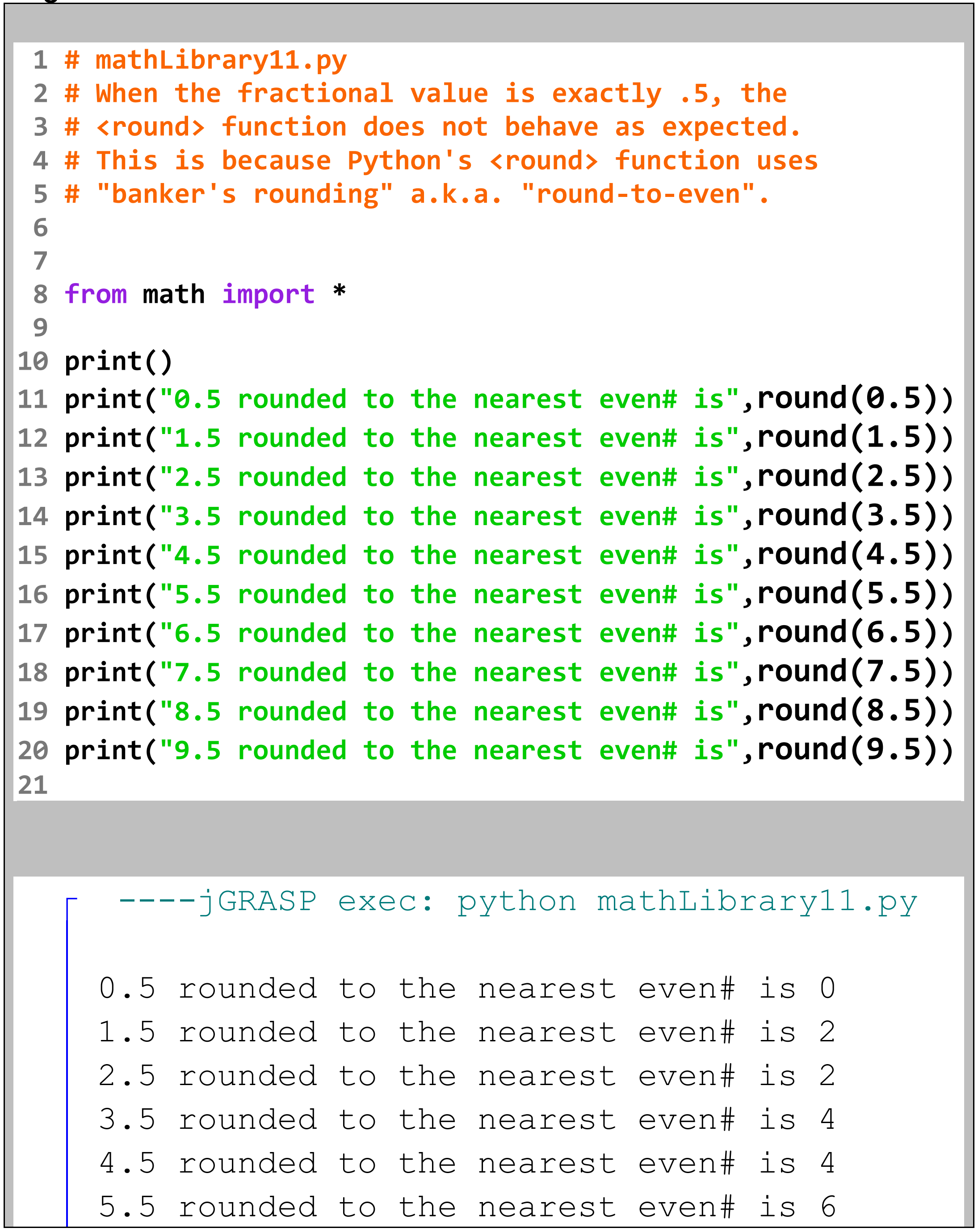
```python
# mathLibrary10.py
# This program demonstrates the <round> function
# which seems to round "normally".


from math import *

print()
print('5.999 rounded "normally" is',round(5.999))
print('5.501 rounded "normally" is',round(5.501))
print('5.5   rounded "normally" is',round(5.5))
print('5.499 rounded "normally" is',round(5.499))
print('5.001 rounded "normally" is',round(5.001))

```

```
 ----jGRASP exec: python mathLibrary10.py

5.999 rounded "normally" is 6
5.501 rounded "normally" is 6
5.5   rounded "normally" is 6
5.499 rounded "normally" is 5
5.001 rounded "normally" is 5

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

Did you notice that I said **round** “seems to round normally?” By “normally” I mean that any number with a decimal part that is less than **.5** rounds down and any number with a decimal part that is greater than or equal to **.5** rounds up. There are some people in the world who do not like this type of rounding because when the decimal part is exactly in the middle, it always rounds up. This causes a slight *bias* which statisticians do not like. They prefer a system where you round up half of the time and round down the other half. One of these techniques is called *Banker’s Rounding* or *Round-To-Even*. Instead of always “rounding up”, when the decimal part of the number is exactly **.5**, this type of rounding will round to the nearest even number.

So why the long explanation on *Banker's Rounding* / *Round-To-Even*? It is because Python's **round** function rounds in this manner. If you need proof, look at program **mathLibrary11.py** and its output, in figure 6.12.

Figure 6.12



```python
# mathLibrary11.py
# When the fractional value is exactly .5, the
# <round> function does not behave as expected.
# This is because Python's <round> function uses
# "banker's rounding" a.k.a. "round-to-even".


from math import *

print()
print("0.5 rounded to the nearest even# is",round(0.5))
print("1.5 rounded to the nearest even# is",round(1.5))
print("2.5 rounded to the nearest even# is",round(2.5))
print("3.5 rounded to the nearest even# is",round(3.5))
print("4.5 rounded to the nearest even# is",round(4.5))
print("5.5 rounded to the nearest even# is",round(5.5))
print("6.5 rounded to the nearest even# is",round(6.5))
print("7.5 rounded to the nearest even# is",round(7.5))
print("8.5 rounded to the nearest even# is",round(8.5))
print("9.5 rounded to the nearest even# is",round(9.5))
```

```
 ----jGRASP exec: python mathLibrary11.py

 0.5 rounded to the nearest even# is 0
 1.5 rounded to the nearest even# is 2
 2.5 rounded to the nearest even# is 2
 3.5 rounded to the nearest even# is 4
 4.5 rounded to the nearest even# is 4
 5.5 rounded to the nearest even# is 6
```

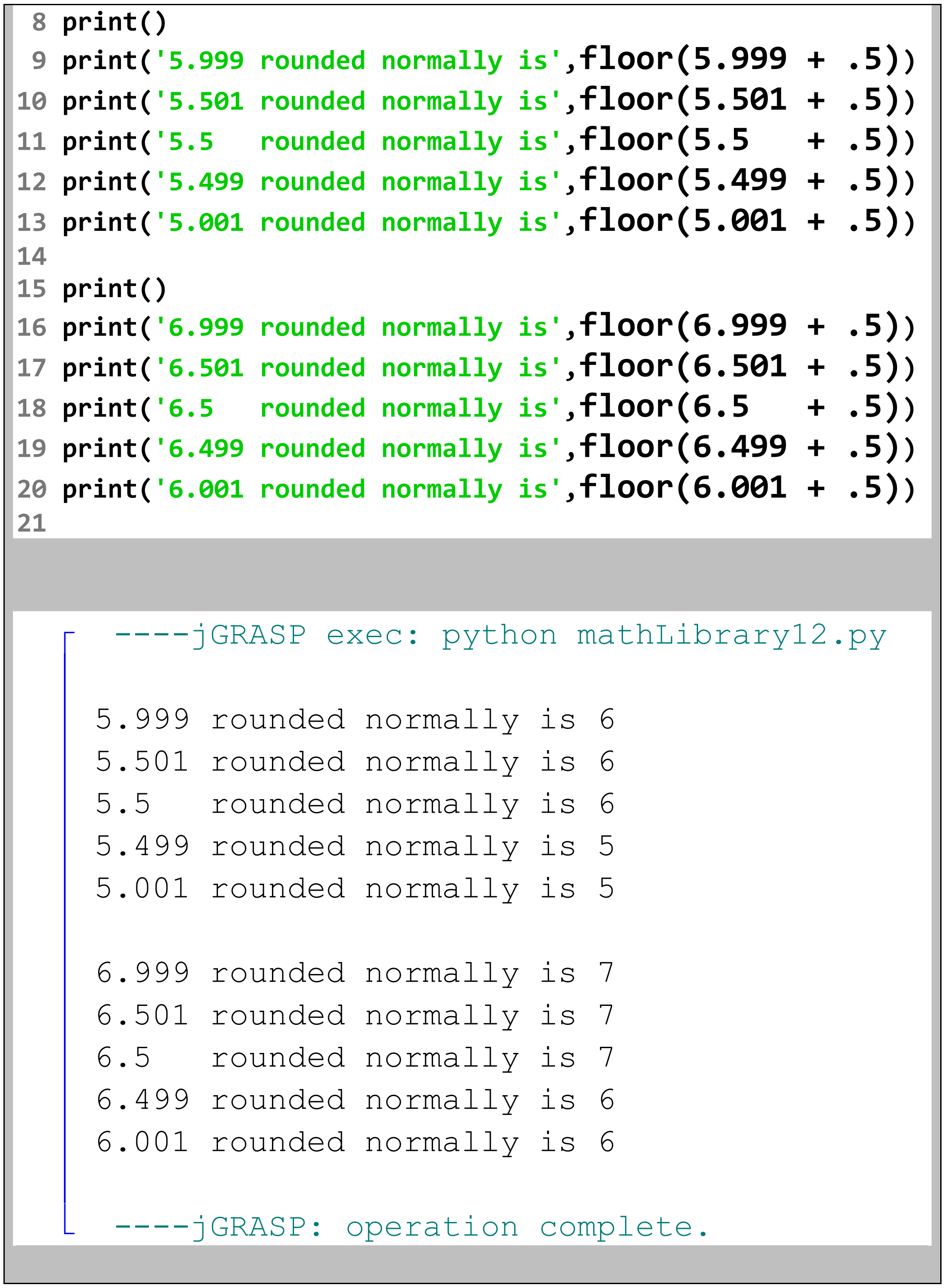
```
6.5 rounded to the nearest even# is 6
7.5 rounded to the nearest even# is 8
8.5 rounded to the nearest even# is 8
9.5 rounded to the nearest even# is 10

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

So if you are a banker or a statistician, Python's **round** function is perfect. However, what if you are a teacher? My students at John Paul II High School take their tests online using some software which I created. There is a crucial part in the process where a student clicks "Submit" after finishing his/her test. The webserver checks how many of the student's answers are correct. After that the number correct is divided by the total number of questions on the test, multiplied by **100** and we now have the student's test grade. This is simple enough if the test has 100 or 50 or 25 questions. In other works, if the number of questions is a *factor* of 100. When things divide evenly, there is no problem. However, most of my tests have 60, 70 or 80 questions. This means you could test a test grade which a decimal portion. Something like **69.5**. Now, if you had a grade of **69.5** on a test, you certainly would want it to round up to **70**. The **round** function would actually do this because **70** is an even number. However, what if you has a **70.5** on the test. You would want this to round to **71**; however, Python's **round** function will also round this to **70**. The point I am trying to make is while bankers and statisticians may like the "round to even" method, there are times when "normal rounding" is desirable as well. So how do we accomplish that?

It turns out Python does not have a function that performs "normal rounding"; however, we can actually achieve "normal rounding" with the **floor** function if we do one thing, we just need to add **.5** to the argument value. This is demonstrated by program **mathLibrary12.py**, shown in figure 6.13. If you doubt that this works, think of any real number, add **.5** and then **floor** it – meaning round down. The result will be the nearest integer.

Figure 6.13

```
# mathLibrary12.py
# The secret to "normal rounding" is to use the
# <floor> function and add .5 to the argument.


from math import *

```



```
print()
print('5.999 rounded normally is',floor(5.999 + .5))
print('5.501 rounded normally is',floor(5.501 + .5))
print('5.5   rounded normally is',floor(5.5   + .5))
print('5.499 rounded normally is',floor(5.499 + .5))
print('5.001 rounded normally is',floor(5.001 + .5))

print()
print('6.999 rounded normally is',floor(6.999 + .5))
print('6.501 rounded normally is',floor(6.501 + .5))
print('6.5   rounded normally is',floor(6.5   + .5))
print('6.499 rounded normally is',floor(6.499 + .5))
print('6.001 rounded normally is',floor(6.001 + .5))

```

```
 ----jGRASP exec: python mathLibrary12.py

5.999 rounded normally is 6
5.501 rounded normally is 6
5.5   rounded normally is 6
5.499 rounded normally is 5
5.001 rounded normally is 5

6.999 rounded normally is 7
6.501 rounded normally is 7
6.5   rounded normally is 7
6.499 rounded normally is 6
6.001 rounded normally is 6

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

The last of these "rounding" functions is **trunc** which is short for "truncate". To "truncate" something mean to chop off the end. In the case of a real number, "truncating" means to chop off the decimal portion. This may sound a lot like the

floor function which always rounds down. Is this another case of redundancy? Do **trunc** and **floor** do the exact same thing? They are similar, especially when the argument is positive; however, what if the argument is negative? Program **mathLibrary13.py**, in figure 6.14, compares the output of **trunc** and **floor** with both positive and negative arguments. While the results are the same for positive numbers, they are different when the arguments are negative.

Figure 6.14

```python
# mathLibrary13.py
# The program demonstrates the <trunc> function
# which "chops-off" or "truncates" the fractional
# part of a real number.  While this may seem
# identical to the <floor> function, it does
# behave differently with negative numbers.


from math import *

print()
print("5.678 rounded down is",floor(5.678))
print('5.678 "truncated" is ',trunc(5.678))

print()
print("-5.678 rounded down is",floor(-5.678))
print('-5.678 "truncated" is ',trunc(-5.678))

```

```
 ----jGRASP exec: python mathLibrary13.py

5.678 rounded down is 5
5.678 "truncated" is  5

-5.678 rounded down is -6
-5.678 "truncated" is  -5

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

In your math class, you may have learned about the *factorial* function. To computer the factorial of a number, you multiply all of the numbers from 1 to that number. So the factorial of **n**, often written with the notation **n!**, would be

```
1 * 2 * 3 * … * (n – 2) * (n – 1) * n = n!
```

Python has a **factorial** function in its **math** library. This is demonstrated by program **mathLibrary14.py**, in figure 6.15. The reason I am showing this program is not just to demonstrate the **factorial** function. I also want to show that in Python, integer values can get really big. In some languages, there is a limit of about 2 billion for integer values. For some, it is as low as **32,767**. Python does not really have a limit, as you can see by the output of **40!**

Figure 6.15

```python
# mathLibrary14.py
# The program demonstrates the <factorial>
# function of the <math> library which returns
# the mathematical factorial of its argument.

# Example: factorial(n) = n * (n-1) * ... * 2 * 1

# NOTE: This program also demonstrates that in
#       Python integer values can be VERY large.


from math import *

print()
print("5! is",factorial(5))
print()
print("10! is",factorial(10))
print()
print("40! is",factorial(40))

```

```
 ┌  ----jGRASP exec: python mathLibrary14.py
 │
 │ 5! is 120
 │
 │ 10! is 3628800
 │
 │ 40! is 8159152832478977343456112695961158
94272000000000
 │
 └  ----jGRASP: operation complete.
```

Math Values pi & e

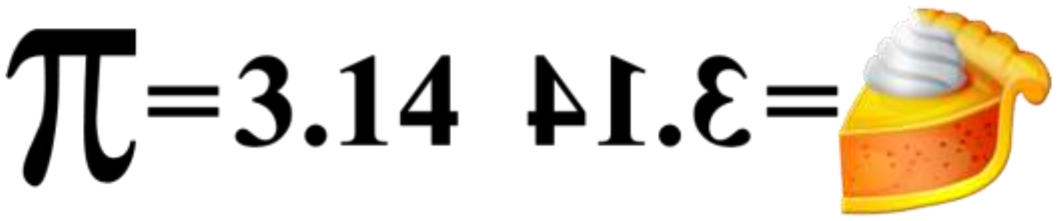
Libraries do not just store functions and procedures. They can also store values. The **math** library stores 2 very important mathematical values. These are the values of **pi** and **e**. By now, you should be familiar with the value of **pi** (π = 3.14159…). If you have not yet learned of the value of **e** in your math class, then simply saying "it is the base of the natural log" probably will not help. For now, this is what you need to know. **e** is similar to **pi** in 2 ways. First, **e** is another non-repeating decimal that goes on forever. (In other words, it is an *irrational number*.) Second, **e** has a special meaning in mathematics and is used in several calculations.

Program **mathLibrary15.py**, in figure 6.16, displays both the value of **pi** and **e** from the **math** library. Notice that **pi** and **e** do not have any arguments. They do not even have any parentheses. That is because **pi** and **e** are not functions, or even procedures for that matter. The output shows that both **pi** and **e** are displayed to 15 decimal places. This is much more accurate than simply using **3.14** or **22 / 7**. From now on, for the remainder of the school year, anytime we need the value of π, we will use **pi** from the **math** library.

Figure 6.16

```python
# mathLibrary15.py
# The <math> library also stores 2 important
# values.  These are the value of <pi> and
# the value of <e>.


from math import *

print()
print("Circumference / Diameter =",pi)
print()
print("Base of the natural log is",e);

```

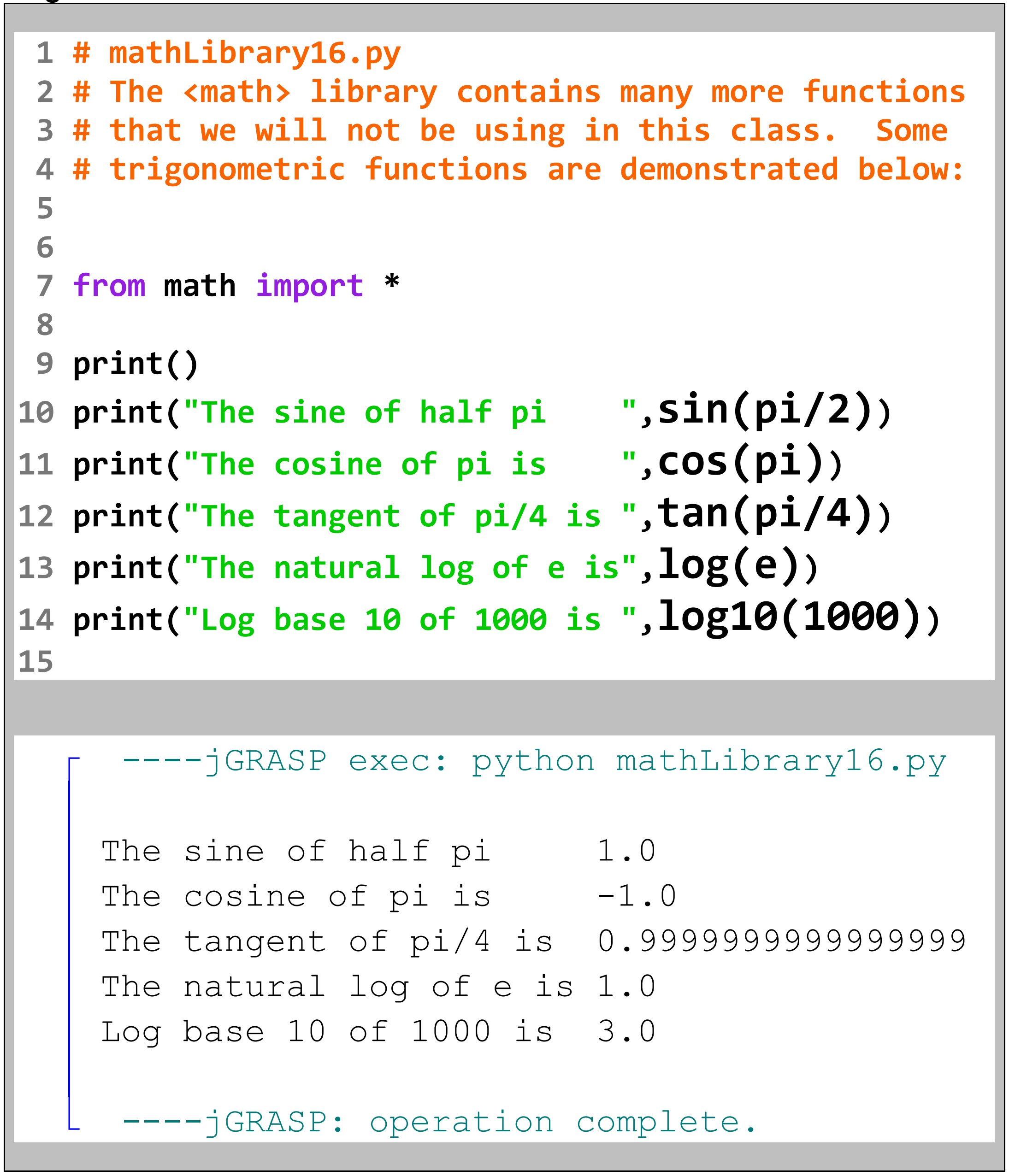
```
 ----jGRASP exec: python mathLibrary15.py

Circumference / Diameter = 3.141592653589793

Base of the natural log is 2.718281828459045

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



Some Trigonometric Functions

The **math** library has many, many different functions to compute a large assortment of things mathematical. We will not even attempt to look at all of them. Instead, we will close this **math** library section with a program that demonstrates a few *trigonometric* functions. Program **mathLibrary16.py**, in figure 6.17, demonstrates functions for *sine* (**sin**), *cosine* (**cos**), *tangent* (**tan**), *natural log* (**log**) and the *log base 10* (**log10**). Note how the values of **pi** and **e** are used as arguments for most of these functions.

Figure 6.17



```python
# mathLibrary16.py
# The <math> library contains many more functions
# that we will not be using in this class.  Some
# trigonometric functions are demonstrated below:


from math import *

print()
print("The sine of half pi    ",sin(pi/2))
print("The cosine of pi is    ",cos(pi))
print("The tangent of pi/4 is ",tan(pi/4))
print("The natural log of e is",log(e))
print("Log base 10 of 1000 is ",log10(1000))

```

```
 ----jGRASP exec: python mathLibrary16.py

The sine of half pi     1.0
The cosine of pi is     -1.0
The tangent of pi/4 is  0.9999999999999999
The natural log of e is 1.0
Log base 10 of 1000 is  3.0

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

Python math Library Disclaimer

Not every function that is *mathematical* in nature is actually part of the **math** library. There are a couple functions, like **round** and **abs**, that can be used without importing math or any other library.

For the sake of simplicity, in this first year class, we will make no attempt to differentiate the *mathematical* functions that require importing the **math** library from the ones that do not.

Instead, we will simply import the **math** library any time we use any function that is *mathematical* in nature… which does not hurt anything.

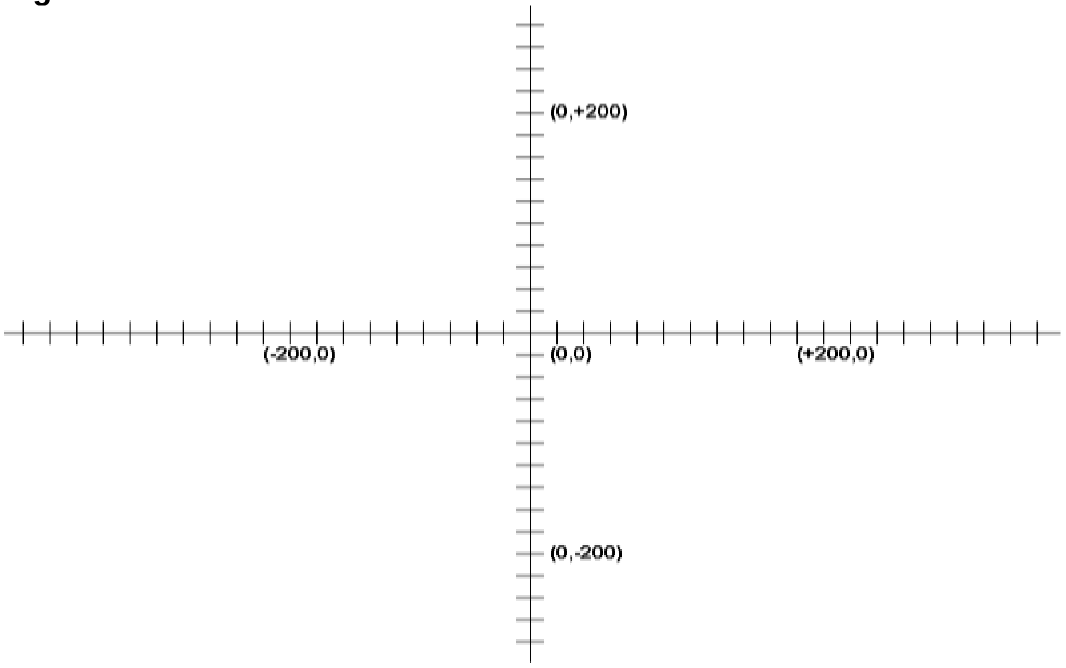
6.4 Introduction to Graphics without the Turtle

In chapter V, you were introduced to *Turtle Graphics*. This first taste of graphics showed you that computer programs do not need to just produce boring text output. The output can be visual and colorful. Since I know that many, if not most, of my students are "visual learners", I want to incorporate as much graphics as possible in this textbook.

Does this mean we are going to learn more *Turtle Graphics*? Actually, no. *Turtle Graphics* was used to introduce graphics programming because it is simplistic. When you start making bigger graphics projects, you may find that *Turtle Graphics* can become a bit tedious. On top of that, when you execute the program, the "turtle" itself is rather slow.

Most programming languages use a form of graphics that is not *turtle-based*. Instead, the graphics in these languages are *coordinate-based*. This is a more efficient way to create graphics output and it is based on something called the *Cartesian Coordinate System*. "What is that?" you may ask. It is probably something with which you are very familiar. Look at figure 6.18. You see a grid with an X-Axis, Y-Axis and 4 quadrants. Every point in the grid is a *coordinate* composed of an X value and a Y value. The **(0,0)** coordinate, otherwise known as "the origin" is located in the center of the grid.

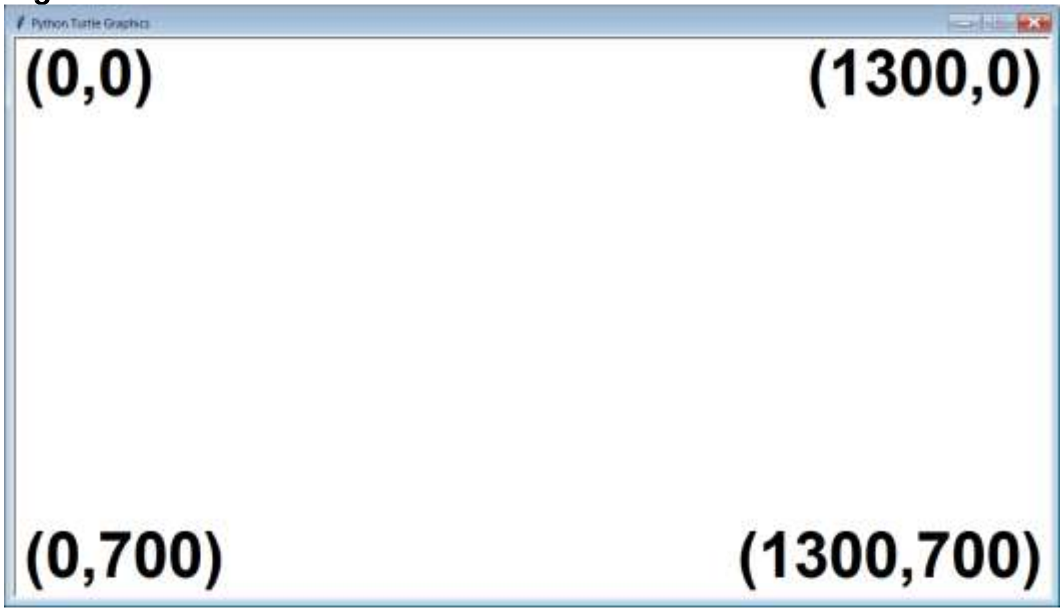
Figure 6.18



Now that we have review *Coordinate Geometry*, you need to realize that traditional computer graphics uses a coordinate system that is very similar to the Cartesian system. Figure 6.19 shows a graphics window dimensioned to **1300 x 700** pixels. This will be the dimensions of most of our graphics windows throughout the year. If your monitor is set to **1366 x 768** pixels, the graphics window will almost take up the entire computer screen. On the other hand, if it is set to something like **1920 x 1080** pixels, the applet window will appear relatively smaller. If you have a new *Ultra-HD* monitor, the window will be downright tiny.

The coordinate values at the four corners of the *graphics window* are displayed. When compared to the *Cartesian* grid there are two significant differences to observe. The **(0,0)** coordinate is located in the top-left-hand corner of the graphics window instead of in the middle. It behaves as if it were just one of the four quadrants of the Cartesian system. The second difference is with the behavior of the Y coordinate values. In a Cartesian system Y-values increase from the bottom up to the top. In a computer graphics window Y-values increase from the top down to the bottom. The X-values in both coordinate systems increase from left to right.

Figure 6.19



Actually, right now the biggest reason for using graphics examples is not that it is more interesting. Graphics procedures use many arguments and working with graphics procedures helps to better understand how to properly use arguments. Functions of the **math** library are fine, but most of them just have a single argument and only a few have 2 arguments. You will see that many of the procedures of the **Graphics** library have several arguments.

Mr. Schram's Graphics Library

You may have noticed that I name dropped the **Graphics** library. Yes, this is the library that we will need to import to do our traditional, coordinate-based graphics programs. There is just one problem. Python does have a not have a **Graphics** library.

You may have been suspecting that Mr. Schram is a little crazy. Now those suspicions seem to be confirmed. How are we going to import a library that does not exist? Clarification: I never said the **Graphics** library does not exist. I simply said Python does not have one. So I simply made my own. In Chapter IX, you will not only learn how to create your own subroutines, you will also learn to store them in your own libraries. For now, you are going to use subroutines from libraries that have been created for you. It does not matter if these libraries are part of the Python programming language, or something that was created by your computer science teacher. The subroutines are still imported and used in exactly the same manner.

Python “Traditional Graphics” Disclaimer

There actually is an interface called **tkinter** than can be used with Python to create graphics output with the traditional coordinate-based system, but that requires some knowledge of *Object Oriented Programming*, which is a topic that is beyond the scope of this first year Computer Science class.

“OOP” is actually a major component of AP® Computer Science-A, which some of you might be taking next year.

Learning Graphics Programming in Intro Computer Science

Learning graphics programming is not simply a fun issue.

Unlike a text-based program, graphics programs are visual.

When you make changes to your program, you see the effects of those changes instantly.

You can learn many sophisticated computer science concepts by manipulating graphics programs.

In order to write programs using traditional, *coordinate-based* graphics, without using *Object Oriented Programming*, we will import Mr. Schram’s **Graphics** library.

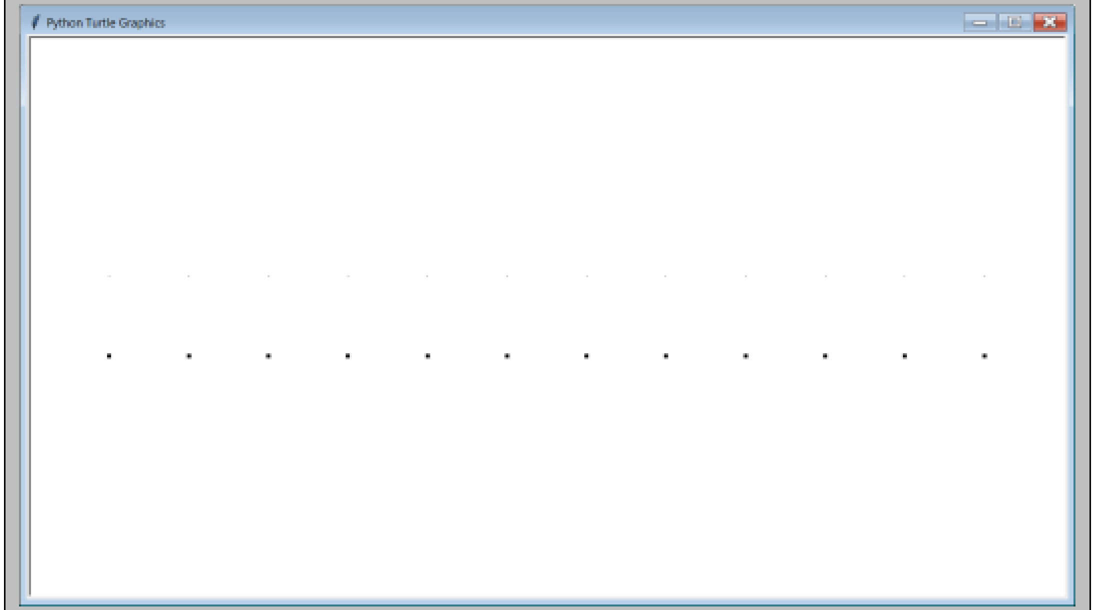
For now, you will be using subroutines from libraries that have been created by others. In Chapter X, you will learn how to create your own subroutines and libraries.

Drawing Pixels and Points

Program **GraphicsLibrary01.py**, in Figure 6.20, will demonstrates both the setup of a traditional *coordinate-based* graphics program and how to draw dots in 2 different sizes. We start by importing the **Graphics** library in the same manner as we have imported other libraries earlier. We then call the **beginGrfx** procedure. This procedure sets up the graphics window and sizes it to our requested dimensions of **(1300,700)**. Then we have a dozen **drawPixel** procedure calls. Each of these has 2 arguments, specifying the X and Y value of the single *pixel* that it will draw. A *pixel* is the tiniest possible dot that can be displayed on a computer screen. Even if you do not have an *Ultra-HD* monitor, pixels can be difficult to see. After that, there are a dozen calls to procedure **drawPoint**. **drawPoint** is similar to **drawPixel**. Both use an X and Y value as arguments. The difference is **drawPoint** draws a small 5 x 5 solid square, instead of a single pixel, which makes it much easier to see. The program finishes with a call to the **endGrfx** procedure. This updates the screen and keeps it open until you are ready to close it.

Figure 6.20

```
# GraphicsLibrary01.py
# This program demonstrates the <drawPixel> and
# <drawPoint> procedures of the <Graphics> library.
# Both procedures draw a dot on the computer screen.

# NOTE: The <Graphics> library was created by Mr. Schram
#       and is not part of standard Python.

# NOTE: Most of the procedures in the <Graphics> library
#       use integer arguments.


# Required to have access to the
# <Graphics> library commands.
from Graphics import *

# Opens a graphics window
# with specified dimensions
beginGrfx(1300,700)

# Draws tiny individual pixels
drawPixel(100,300)
drawPixel(200,300)
drawPixel(300,300)
drawPixel(400,300)
drawPixel(500,300)
drawPixel(600,300)
drawPixel(700,300)
drawPixel(800,300)
drawPixel(900,300)
drawPixel(1000,300)
drawPixel(1100,300)
drawPixel(1200,300)

# Draws small squares
drawPoint(100,400)
drawPoint(200,400)
drawPoint(300,400)
drawPoint(400,400)
drawPoint(500,400)
drawPoint(600,400)
drawPoint(700,400)
drawPoint(800,400)
drawPoint(900,400)
drawPoint(1000,400)
drawPoint(1100,400)
drawPoint(1200,400)

# Updates the screen and keeps the graphics
# window open when the program is finished.
endGrfx()

```



When you look at the output in Figure 6.19, you might have a difficult time seeing the pixels that were plotted by the **drawPixel** commands. There are 12 evenly spaced pixels that are just above the 12 evenly spaced small squares. If you do not see them, then simply trust that they are there. Part of what makes modern digital images so sharp is that they are made up of millions of tiny pixels. The smaller the pixels, the sharper the image becomes. This might explain why it is hard to see just one.

6.5 Drawing Simple Shapes

A large portion of the procedures in the **Graphics** library involve drawing various shapes. We will start with some of the simpler shapes line lines, rectangles, circles, ovals and arcs.

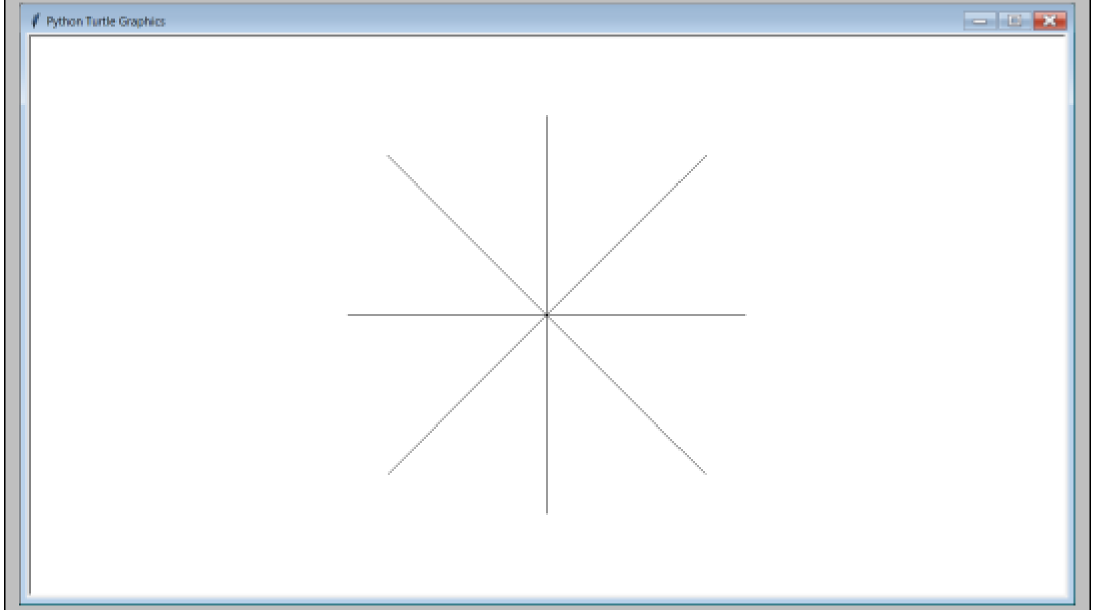
Drawing Lines

Procedure **drawLine** draws straight *line segments*, which start at one point and finish at another. This requires a total of 4 integer arguments. The first 2 integers specify the starting coordinate. The third and fourth integers specify the ending coordinate. Program **GraphicsLibrary02.py**, in Figure 6.21, demonstrates using **drawLine** to draw a large snowflake comprised of 4 different lines.

Figure 6.21

```
# GraphicsLibrary02.py
# This program demonstrates the <drawLine>
# procedure of the <Graphics> library.
# Lines are drawn from (x1,y1) to (x2,y2)
# with <drawLine(x1,y1,x2,y2)>.
# This program displays a snowflake by
# calling <drawLine> 4 times.

```

```
from Graphics import *

beginGrfx(1300,700)

drawLine(650,100,650,600)
drawLine(400,350,900,350)
drawLine(450,150,850,550)
drawLine(850,150,450,550)

endGrfx()
```



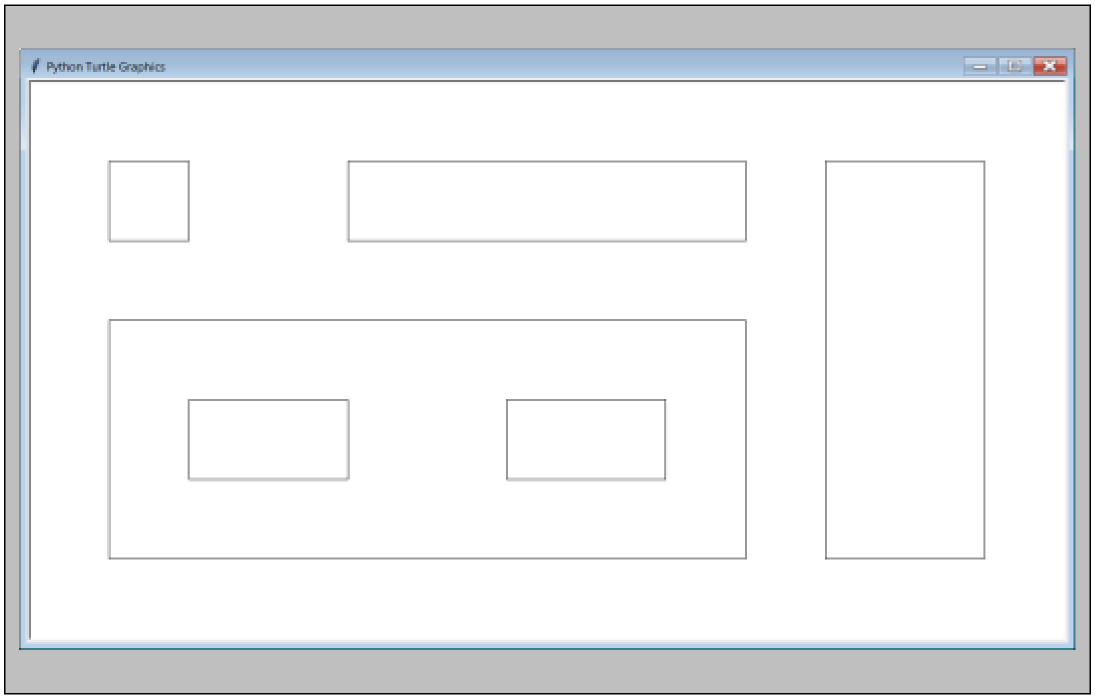
An excellent learning technique during a lab/lecture is to "mess with the programs". Change them in some way and see what happens. Try to add one or more lines to the output. You can also try to "move" the snowflake.

Drawing Rectangles

A line is one-dimensional; it only has length. Many of the graphics procedures of the **Graphics** library are two-dimensional and have the option to draw different shapes. **drawRectangle** is demonstrated in program **GraphicsLibrary03.py**, shown in Figure 6.22. The arguments for **drawRectangle** are actually the same as **drawLine** with one important difference. In **drawLine**, the 2 coordinate points you specify (as in X1, Y1, X2, Y2) are the starting point and the ending point of the line. In **drawRectangle**, you also specify 2 coordinate points, but they represent the upper-left-hand coordinate (X1,Y1) and the lower-right-hand coordinate (X2,Y2) of the rectangle.

Figure 6.22

```
# GraphicsLibrary03.py
# This program demonstrates the <drawRectangle>
# procedure of the <Graphics> library.
# Rectangles are drawn from the upper-left-hand
# corner(x1,y1) to the lower-right-hand corner
# (x2,x2) with <drawRectangle(x1,y1,x2,y2)>.


from Graphics import *

beginGrfx(1300,700)

drawRectangle(100,100,200,200)
drawRectangle(400,100,900,200)
drawRectangle(100,300,900,600)
drawRectangle(1000,100,1200,600)
drawRectangle(200,400,400,500)
drawRectangle(600,400,800,500)

endGrfx()

```

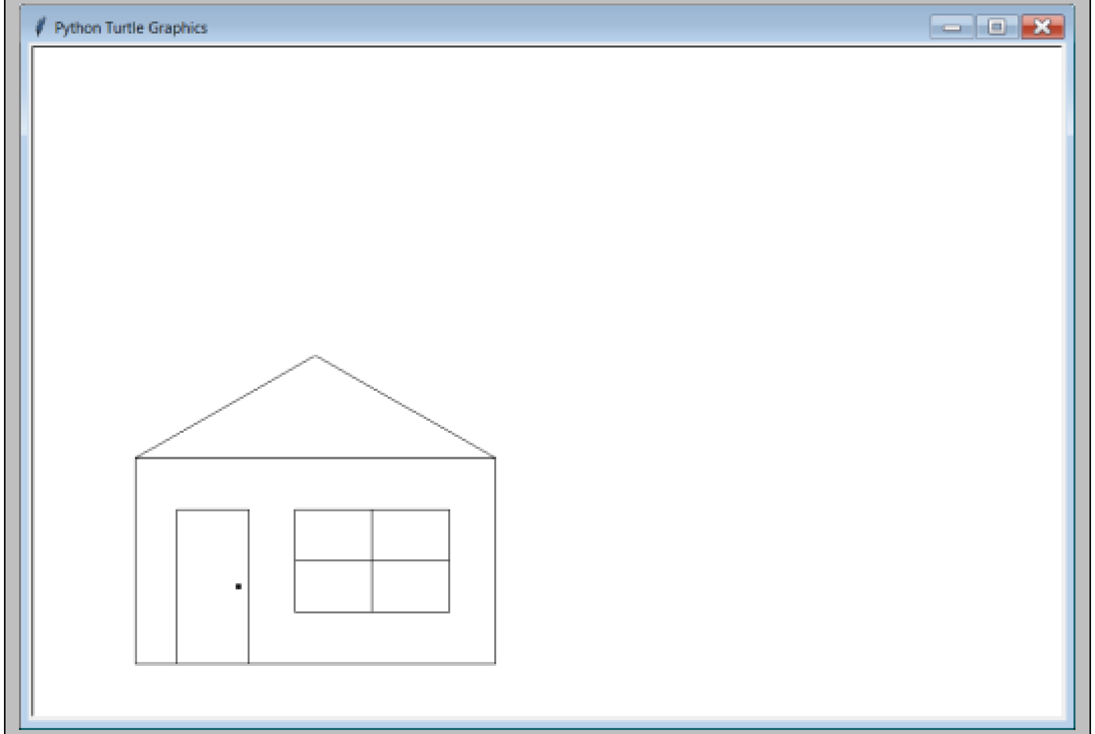


Combining Points, Lines and Rectangles

You may not realize it, but you already know enough graphics commands to draw some simple images. Program **GraphicsLibrary04.py**, in Figure 6.23, combines 3 rectangles, 4 lines and 1 point to create a simple house.

Figure 6.23

```
# GraphicsLibrary04.py
# This program combines lines, rectangles
# and a point to draw a simple house.


from Graphics import *

beginGrfx(1000,650)

drawRectangle(100,400,450,600)
drawLine(100,400,275,300)
drawLine(450,400,275,300)
drawRectangle(255,450,405,550)
drawLine(330,450,330,550)
drawLine(255,500,405,500)
drawRectangle(140,450,210,600)
drawPoint(200,525)

endGrfx()

```

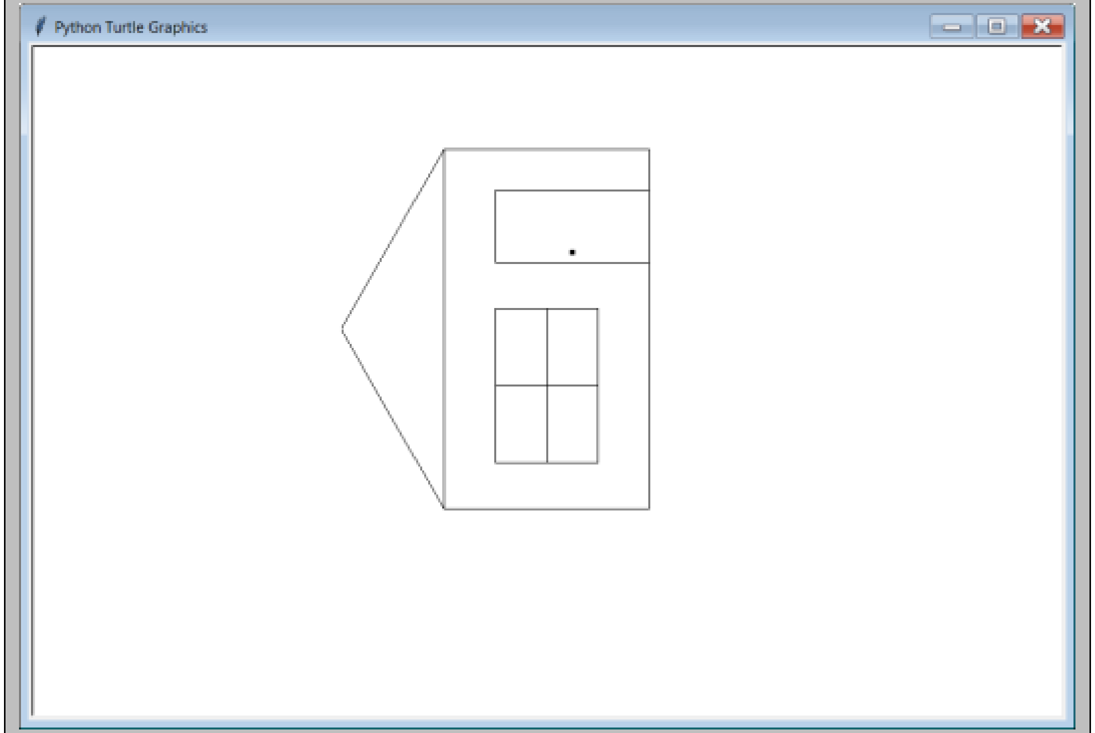


Common Logic Error: Switching X and Y

Program **GraphicsLibrary05.py**, in Figure 6.24, is very similar to the previous program. It still has 3 rectangles, 4 lines and 1 point. It still has all of the same integers. The program still executes, but the output is not what we want. This is a perfect example of a *Logic Error*. In this case, the *error* is that all of the X and Y values in the program are switched. The result is the house is flipped diagonally.

Figure 6.24

```
# GraphicsLibrary05.py
# This program demonstrates the Logic Error
# that occurs when X and Y values are switched.
# The house is "flipped" diagonally.


from Graphics import *

beginGrfx(1000,650)

drawRectangle(400,100,600,450)
drawLine(400,100,300,275)
drawLine(400,450,300,275)
drawRectangle(450,255,550,405)
drawLine(450,330,550,330)
drawLine(500,255,500,405)
drawRectangle(450,140,600,210)
drawPoint(525,200)

endGrfx()
```

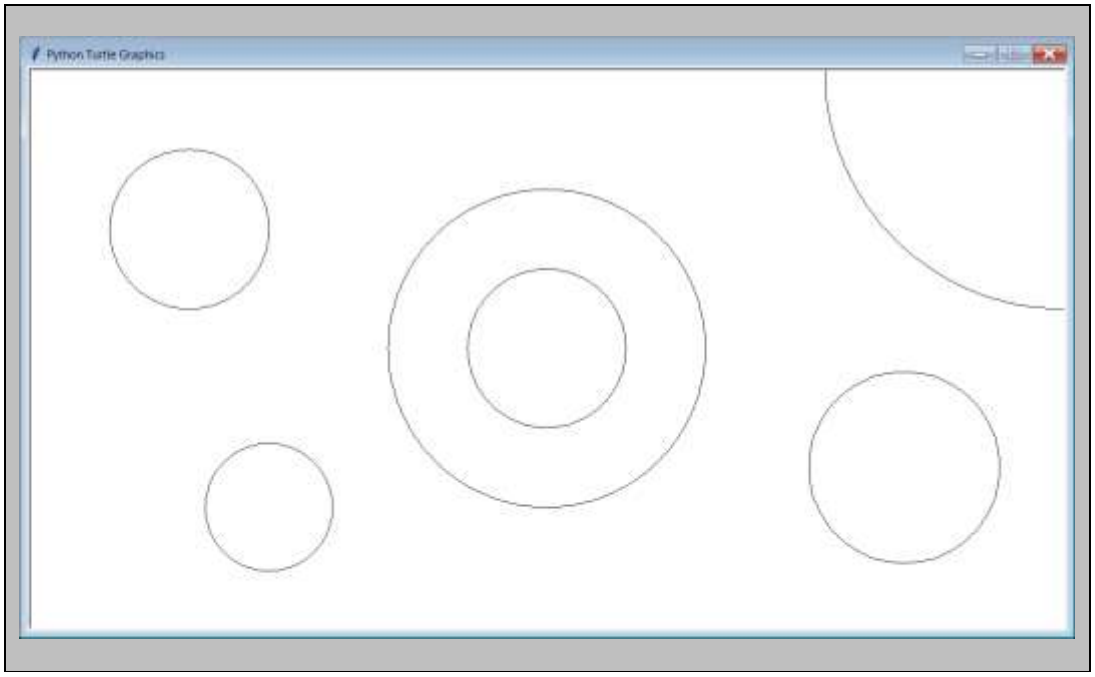


Drawing Circles

The mathematical definition of a circle is *a set of points in a plane that are equidistant from a given point*. The given point is the X and Y coordinate of the center of the circle. The distance from the center of the circle to any point on the edge of the circle itself is called the *radius*. No matter how you draw a line from the circle's center to its edge, the distance will be the same (hence the word "equidistant" earlier). It should come as no surprise that the arguments for *drawCircle* will involve the X and Y coordinate of the center, as well as the radius. Program **GraphicsLibrary06.py**, in Figure 6.25, draws several circles.

Figure 6.25

```
# GraphicsLibrary06.py
# This program demonstrates the <drawCircle>
# procedure of the <Graphics> library.
# Circles are drawn from their center (x,y)
# with a particular radius with
# <drawCircle(x,y,radius)>.


from Graphics import *

beginGrfx(1300,700)

drawCircle(200,200,100)
drawCircle(1300,0,300)
drawCircle(650,350,100)
drawCircle(650,350,200)
drawCircle(300,550,80)
drawCircle(1100,500,120)

endGrfx()

```



Common Syntax Error: Wrong # of Arguments

The **drawCircle** procedure has 3 integer arguments. What happens if you decide to throw in an extra integer, or leave one out? The next couple programs will look at this. Program **GraphicsLibrary07.py**, in Figure 6.26, demonstrates the syntax error that occurs when we try to call **drawCircle** with only 2 arguments.

Figure 6.26

```
# GraphicsLibrary07.py
# This program demonstrates the Syntax Error
# that occurs when a function is called
# with too few arguments.

from Graphics import *

beginGrfx(1300,700)

drawCircle(650,350)

endGrfx()

```

```
 ----jGRASP exec: python GraphicsLibrary07.py
Traceback (most recent call last):
  File "GraphicsLibrary07.py", line 10, in <module>
    drawCircle(650,350)
TypeError: drawCircle() missing 1 required
positional argument: 'r'

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

Program **GraphicsLibrary08.py**, in Figure 6.27, demonstrates the syntax error that occurs when we try to call **drawCircle** with an extra argument.

Figure 6.27

```
# GraphicsLibrary08.py
# This program demonstrates the Syntax Error
# that occurs when a function is called
# with too many arguments.

from Graphics import *

beginGrfx(1300,700)

drawCircle(650,350,100,200)

endGrfx()
```

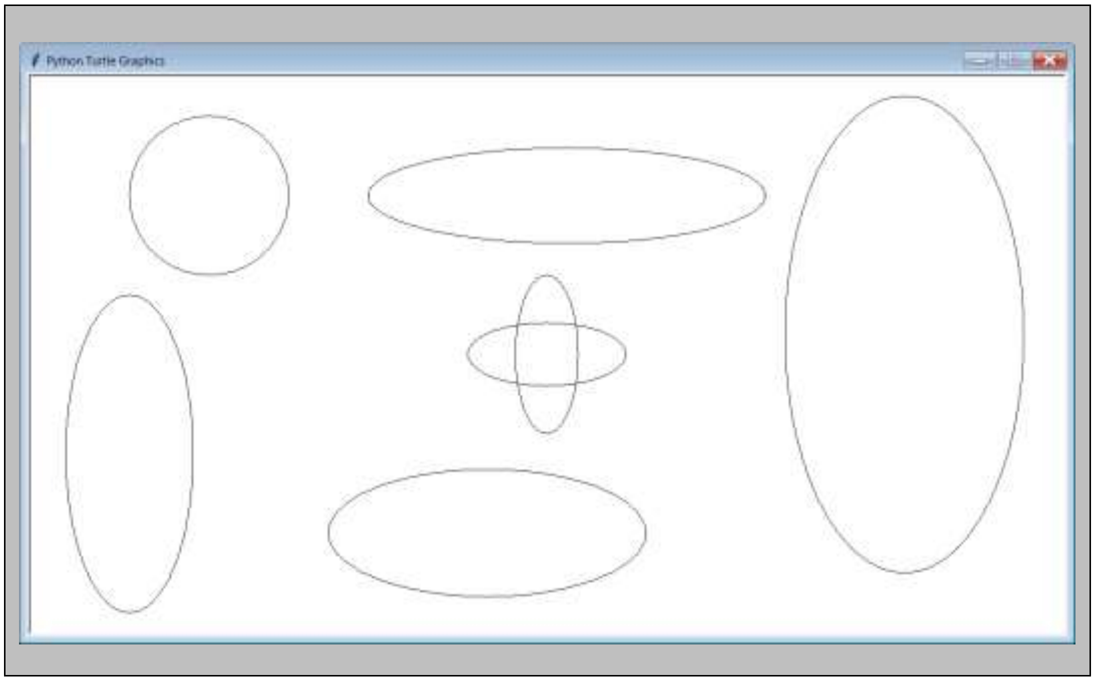
```
 ----jGRASP exec: python GraphicsLibrary08.py
Traceback (most recent call last):
  File "GraphicsLibrary08.py", line 10, in <module>
    drawCircle(650,350,100,200)
TypeError: drawCircle() takes 3 positional
arguments but 4 were given

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

Drawing Ovals

Drawing an oval is very similar to drawing a circle. You still have to specify the X and Y coordinates of the center. What is different is now you do not have one radius. You now have 2 *radii*. There is a *horizontal radius*, and there is a *vertical radius*. Program **GraphicsLibrary09.py**, in Figure 6.28, draws several ovals. Note that when the *horizontal radius* is bigger, you get a short, fat oval. When the *vertical radius* is bigger, you get a tall, thin oval. What about when both *radii* are the same? In that case, you get a circle. This is how circles are drawn in some programming languages which have a **drawOval** procedure, but not a **drawCircle** procedure.

Figure 6.28

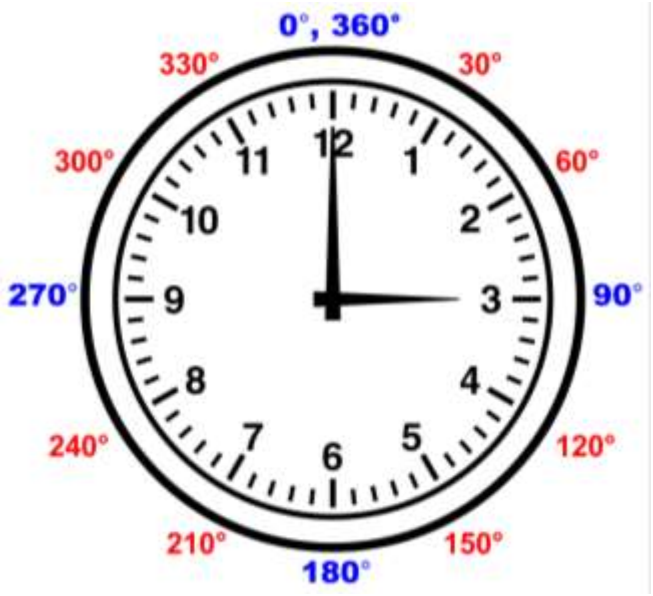
```
# GraphicsLibrary09.py
# This program demonstrates the <drawOval>
# procedure of the <Graphics> library.
# Ovals are drawn from their center (x,y)
# with a horizontal radius (hr) and a vertical
# radius (vr) with <drawOval(x,y,hr,vr)>.


from Graphics import *

beginGrfx(1300,700)

drawOval(225,150,100,100)
drawOval(1100,325,150,300)
drawOval(675,150,250,60)
drawOval(650,350,40,100)
drawOval(650,350,100,40)
drawOval(125,475,80,200)
drawOval(575,575,200,80)

endGrfx()

```



Drawing Arcs

The first thing you need to understand about drawing an arc is *an arc is a piece of an oval*. Because of this, you will notice that the first 4 arguments of **drawArc** are identical to **drawOval**. You still have the X and Y coordinate values of the center, and the 2 radii. For an arc you need 2 additional arguments. To understand what they mean, look at Figure 6.29.

Figure 6.29



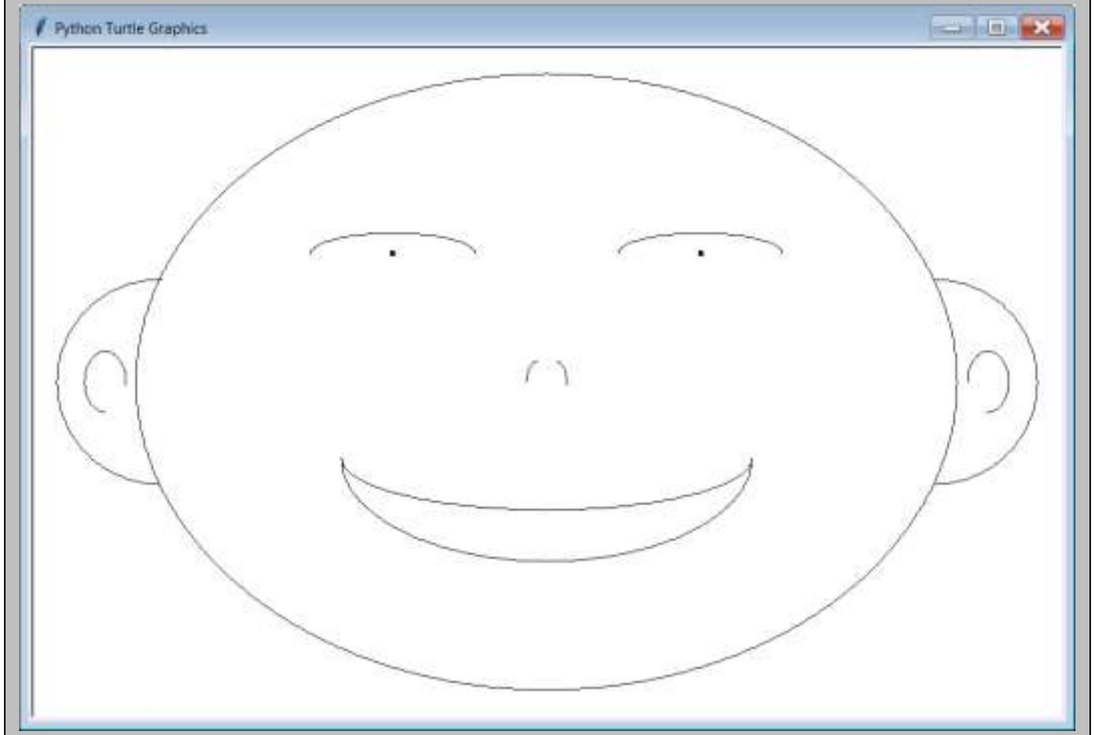
Compare the clock to a circle. In math classes you learned that a circle has 360 degrees. To draw an *arc*, you need to specify the *starting degree value*, and the *stopping degree value*. On a clock, the 0 degree position is at 12:00. The 90 degree position is at 3:00. 180 degrees is at 6:00. 270 degrees is at 9:00. And 360 degrees is back at 12:00 again.

If you want to draw the bottom half of a circle – possibly to draw a smiley face ☺ – you need to use **90** and **270** as the last 2 arguments of the **drawArc** command. This draws a partial oval which starts at 90 degrees (3:00) and goes clockwise to 270 degrees (9:00). If the **90** and **270** are reversed, you get the top half of the oval, because it will start at 270 degrees (9:00) and end at 90 degrees (3:00). This is would be good to use if you are drawing something like an eyebrow.

Figure 6.30 shows program **GraphicsLibrary10.py** which draws several arcs. Unlike several of the previous programs, which just drew several shapes, this program combines the arcs into a picture. With the exception of the 2 points used for the eyes, everything you see in the output is done with an arc. The head may look a lot like an oval, but it is actually an arc that is starting at 0 degrees (12:00) and going 360 degrees which is completely around the oval.

Figure 6.30

```
# GraphicsLibrary10.py
# This program demonstrates the <drawArc> procedure of the
# <Graphics> library.  An "arc" is a piece of an "oval".
# Like ovals, arcs are drawn from their center (x,y) with
# a horizontal radius (hr) and a vertical radius (vr).
# Arcs also require a starting and stopping degree value.
# This is done with <drawArc(x,y,hr,vr,start,stop)>.


from Graphics import *

beginGrfx(1000,650)

drawArc(500,325,400,300,0,360)   # complete oval
drawArc(500,400,200,50,90,270)   # bottom half of an oval
drawArc(500,400,200,100,90,270)
drawArc(350,200,80,20,270,90)    # top half of an oval
drawArc(650,200,80,20,270,90)
drawArc(123,325,100,100,180,0)   # left half of an oval
drawArc(878,325,100,100,0,180)   # right half of an oval
drawArc(490,325,10,20,270,360)   # top-left 1/4 of an oval
drawArc(510,325,10,20,0,90)      # top-right 1/4 of an oval
drawArc(70,325,20,30,180,90)      # 3/4 of an oval
drawArc(930,325,20,30,270,180)   # different 3/4 of an oval
drawPoint(350,200)
drawPoint(650,200)

endGrfx()
```



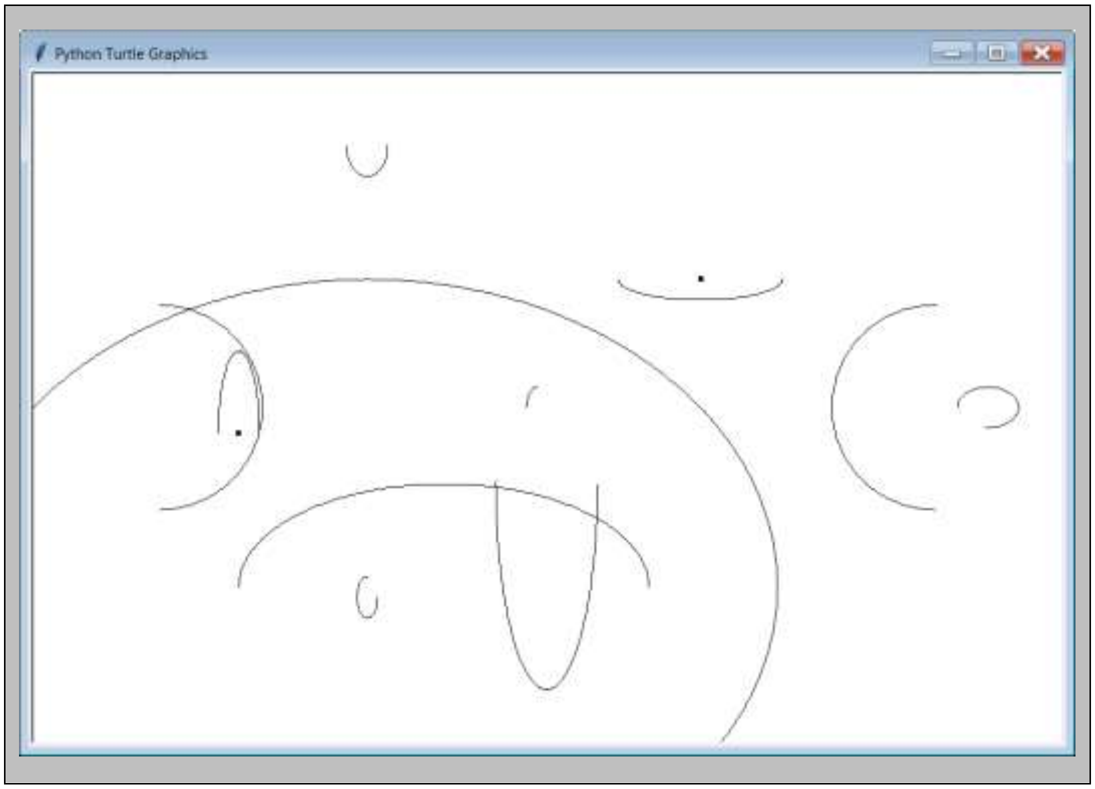
Watch Your Argument Sequence

Earlier, you saw what happens when your X and Y values are switched. Now we have procedures with several arguments. It is not just X and Y. The **drawArc** procedure also has 2 radii, a starting value and an ending value. There are 6 different integer arguments used in **drawArc**. What happens if they all get mixed up?

Program **GraphicsLibrary11.py**, in Figure 6.31, addresses this very issue. At first glance, the program looks just like the previous program. Same commands. Same numbers. The problem is the numbers are in a completely different order. When you see the output, you should understand why it is important to keep your arguments in the proper order.

Figure 6.31

```
# GraphicsLibrary11.py
# This program may seem almost identical to the
# previous program which drew the smiley face.
# In reality, it demonstrates what happens when
# arguments are put in the wrong order.  The
# program does execute without any errors, but
# the results are not what you expect.
# This is another example of a Logic Error.


from Graphics import *

beginGrfx(1000,650)

drawArc(325,500,400,300,0,360)
drawArc(500,400,50,200,90,270)
drawArc(400,500,200,100,270,90)
drawArc(200,350,20,80,270,90)
drawArc(650,200,80,20,90,270)
drawArc(123,325,100,100,0,180)
drawArc(878,325,100,100,180,0)
drawArc(490,325,10,20,270,360)
drawArc(325,510,10,20,90,0)
drawArc(325,70,20,30,90,270)
drawArc(930,325,30,20,270,180)
drawPoint(200,350)
drawPoint(650,200)

endGrfx()

```



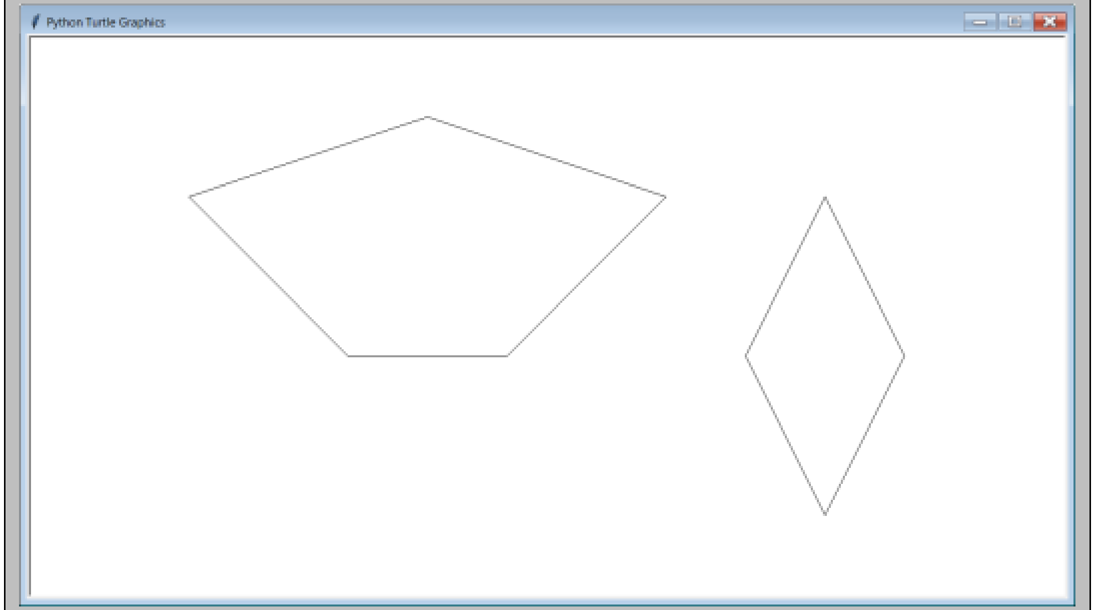
6.6 Drawing Polygons

OK, we can draw some simple shapes, but what if we need to draw something that is not a rectangle, circle or oval? What is we need to draw a polygon with many sides. Technically, this can be done with several **drawLine** commands, but that is tedious. The **Graphics** library has several procedures for drawing different types of *polygons*.

Program **GraphicsLibrary12.py**, in Figure 6.32, demonstrates the **drawPolygon** procedure and how it can be called to display both a pentagon and a triangle. **drawPolygon** is a special king of procedure where the number of arguments seem to be flexible. In this case, I first call the procedure with 10 integer arguments. These are interpreted as 5 coordinate points. The computer literally "connects the dots" and draws a pentagon. Then I call **drawPolygon** again, but with 8 integer arguments. This is interpreted as 4 coordinate points and draws a diamond. You probably noticed the extra sets of brackets **[]**. These do have a purpose, which be explained in a later chapter during the second semester.

Figure 6.32

```
# GraphicsLibrary12.py
# This program demonstrates the <drawPolygon> procedure.
# <drawPolygon> can handle 3 or more sets of coordinate
# points to draw a triangle, quadrilateral, pentagon,
# hexagon, octagon, or any other polygon.
# The purpose of the extra set of brackets [] in the
# procedure call will be explained in a later chapter.


from Graphics import *

beginGrfx(1300,700)

drawPolygon([500,100,800,200,600,400,400,400,200,200])
drawPolygon([900,400,1000,200,1100,400,1000,600])

endGrfx()

```

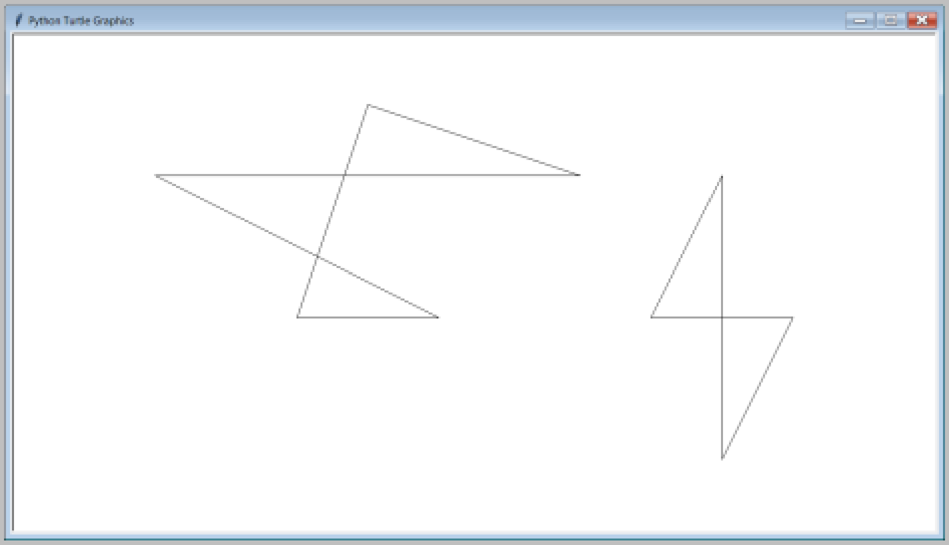


Watch Your Coordinate Pair Sequence

"Coordinate Pair Sequence" is just a special form of "Argument Sequence". Even if every X value is properly followed by its corresponding Y value, it is still possible for these "coordinate pairs" to be out of sequence. Program **GraphicsLibrary13.py**, in Figure 6.33, is pretty much the same program as **GraphicsLibrary12.py**. The difference is that program **GraphicsLibrary13.py** has rearranged the coordinate pairs in both **drawPolygon** commands. Note how this drastically alters the output. Even though the exact same coordinate points are used, they are connected differently because the computer "connects the dots" based on the sequence of coordinate pairs in the procedure call.

Figure 6.33

```
# GraphicsLibrary13.py
# This program demonstrates that the sequence of
# the coordinate pairs is significant.  The same
# coordinates from the previous program are used
# in a different sequence.  The result is that
# the display is now very different.


from Graphics import *

beginGrfx(1300,700)

drawPolygon([400,400,500,100,800,200,200,200,600,400])
drawPolygon([900,400,1000,200,1000,600,1100,400])

endGrfx()
```



Drawing Regular Polygons

A *regular polygon* is a polygon with the properties that all of the sides are the same length and all of the angles have the same measure.

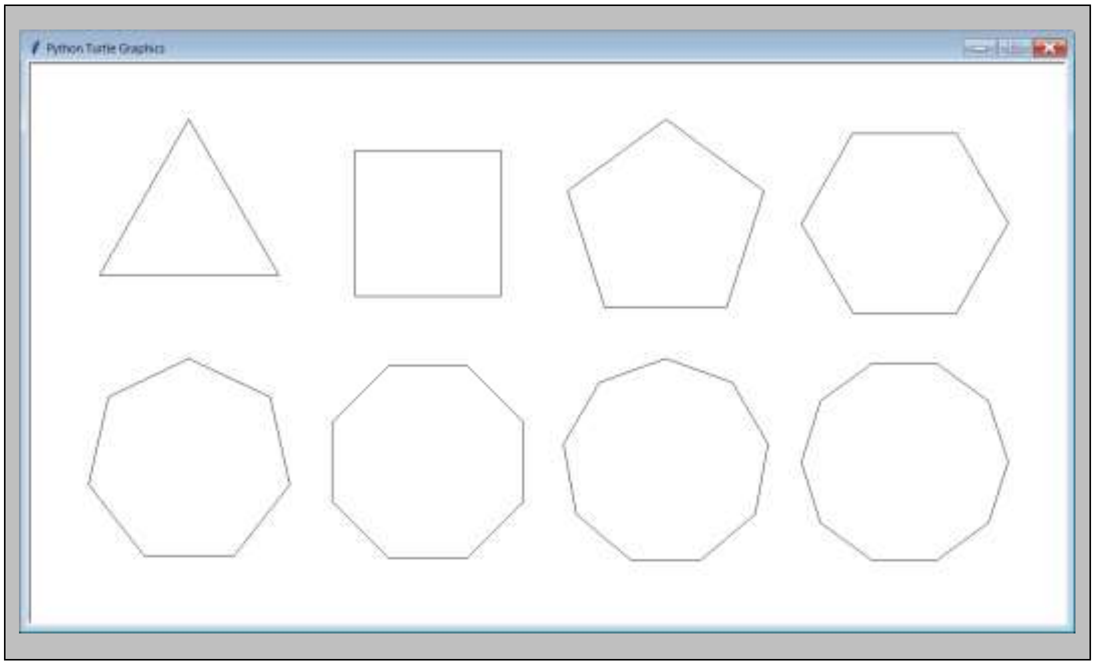
Suppose you want to draw something like a stop sign, or a honeycomb. A stop sign is a regular octagon. A honeycomb is made up of many regular hexagons. Technically, you can use **drawPolygon** to draw *regular* polygons, but that would require either a lot of *trial and error*, or a lot of math – specifically trigonometry.

The **Graphics** library has a few procedures to help make this simpler. The first of these is **drawRegularPolygon**. It has arguments that are similar to **drawCircle**. Both start with the [X,Y] coordinate of the center. After that, there is the radius. Up to this point **drawCircle** and **drawRegularPolygon** are the same, but **drawRegularPolygon** has one additional argument at the end to indicate the number of sides.

Program **GraphicsLibrary14.py**, in Figure 6.34, demonstrates 8 different regular polygons. There is an equilateral triangle (3 sides), square (4 sides), *regular pentagon* (5 sides), *regular hexagon* (6 sides), *regular heptagon* (7 sides), *regular octagon* (8 sides), *regular nonagon* (9 sides) and a *regular decagon* (10 sides).

Figure 6.34

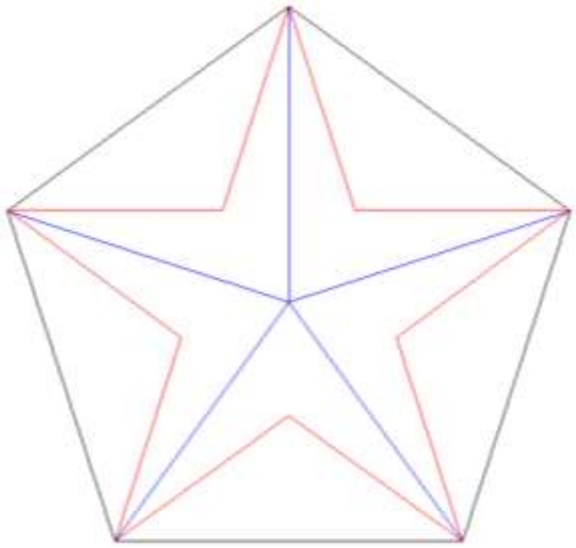
```
# GraphicsLibrary14.py
# This program demonstrates the <drawRegularPolygon>
# procedure of the <Graphics> library.  Regular Polygons
# are drawn from their center (x,y) with a certain radius
# (of the circumscribing circle) and a certain number of
# sides with <drawRegularPolygon(x,y,radius,numSides)>.


from Graphics import *

beginGrfx(1300,700)

drawRegularPolygon(200,200,130,3)
drawRegularPolygon(500,200,130,4)
drawRegularPolygon(800,200,130,5)
drawRegularPolygon(1100,200,130,6)
drawRegularPolygon(200,500,130,7)
drawRegularPolygon(500,500,130,8)
drawRegularPolygon(800,500,130,9)
drawRegularPolygon(1100,500,130,10)

endGrfx()
```



Special Regular Polygons

To me, *stars* and *snowflakes* are special kinds of regular polygons. In Figure 6.35 you can see that a 5-point star and a 5-point snowflake both fit perfectly inside a 5-sided regular polygon. Stars are used in flags and many other pictures. While the *snowflake* design can certainly be used for snowflakes, it also works for things like bicycle spokes and the *bursting* effect of fireworks. For this reason, I will refer to the *snowflake* design as a "burst" for the remainder of this chapter.

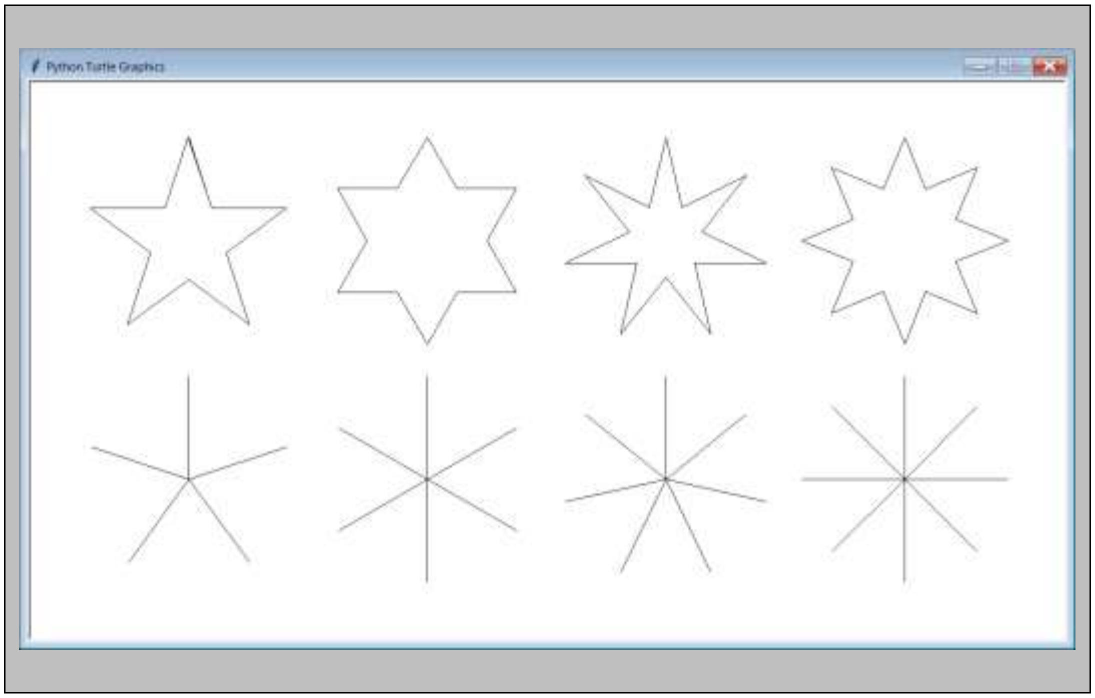
Figure 6.35



The **Graphics** library also has a couple procedures for drawing *stars* and *bursts*. These are **drawStar** and **drawBurst** (surprisingly). Both of these procedures are very similar to **drawRegularPolygon** and the arguments are practically the same. The only difference is that with the **drawPolygon**, the last argument indicates the number of *sides*. With **drawStar**, the last argument indicates the number of *points*. With **drawBurst**, the last argument indicates the number of *lines*. Program **GraphicsLibrary15.py**, in Figure 6.36, demonstrates both **drawStar** and **drawBurst**.

Figure 6.36

```
# GraphicsLibrary15.py
# This program demonstrates the <drawStar> & <drawBurst>
# procedures of the <Graphics> library.  Stars and bursts
# are drawn from their center (x,y) with a certain radius
# (of the circumscribing circle) and a certain number of
# points/lines with <drawStar(x,y,radius,numPoints)>
# and <drawBurst(x,y,radius,numLines)>.


from Graphics import *

beginGrfx(1300,700)

drawStar(200,200,130,5)
drawStar(500,200,130,6)
drawStar(800,200,130,7)
drawStar(1100,200,130,8)
drawBurst(200,500,130,5)
drawBurst(500,500,130,6)
drawBurst(800,500,130,7)
drawBurst(1100,500,130,8)

endGrfx()


```

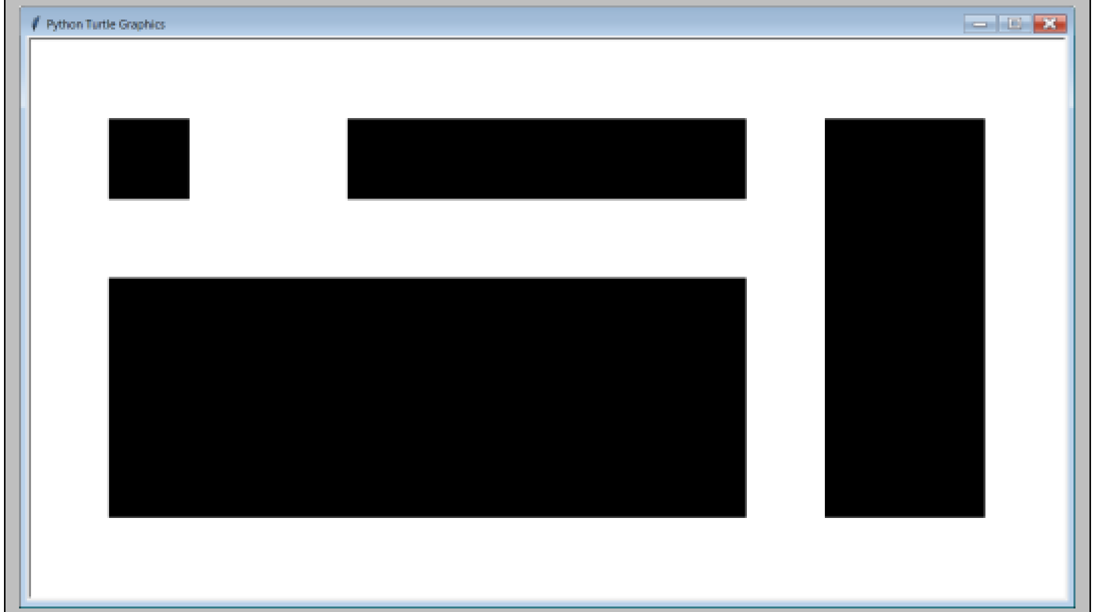


6.7 Fill Procedures and Colors

You have been shown several *draw* procedures of the **Graphics** library. In this section *fill* procedures and graphics colors are introduced. Most of the *drawing* procedures – the ones that draw enclosed shapes – have a corresponding *fill* procedure. The *fill* procedures have the exact same arguments, with the exact same meaning as the corresponding *draw* procedures. For example, program **GraphicsLibrary16.py**, in Figure 6.37, practically repeats the **drawRectangle** example from program **GraphicsLibrary03.py**, which drew 6 *open* rectangles. Program **GraphicsLibrary16.py** simply changes the word **draw** to **fill**. Now 6 *solid* or *filled-in* rectangles are drawn.

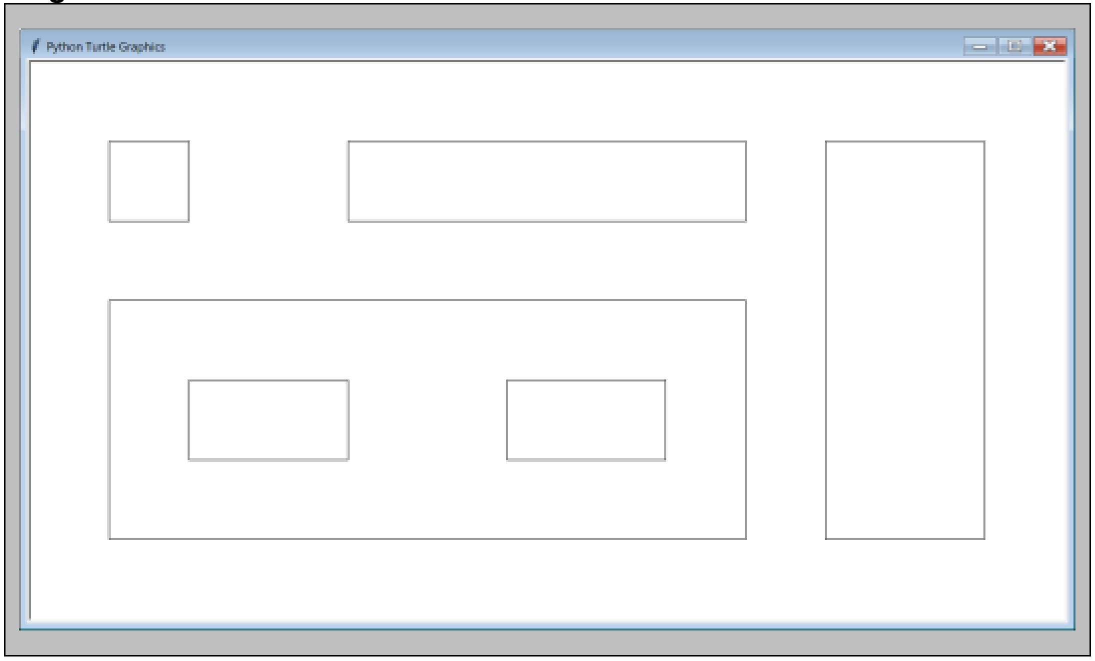
Figure 6.37

```
# GraphicsLibrary16.py
# This program demonstrates the <fillRectangle>
# procedure of the <Graphics> library.  The
# arguments for <fillRectangle> are exactly
# the same as <drawRectangle>.  Even though
# 6 solid rectangles are drawn, only 4 show
# up on the screen.  Where are the other 2?


from Graphics import *

beginGrfx(1300,700)

fillRectangle(100,100,200,200)
fillRectangle(400,100,900,200)
fillRectangle(100,300,900,600)
fillRectangle(1000,100,1200,600)
fillRectangle(200,400,400,500)
fillRectangle(600,400,800,500)

endGrfx()

```



The output of this program has a little mystery to it. Are you surprised to see only 4 rectangles? Where did the other 2 go? Let us look at the output of program **GraphicsLibrary03.py** again. (See Figure 6.38.) You do see 6 rectangles there.

Figure 6.38



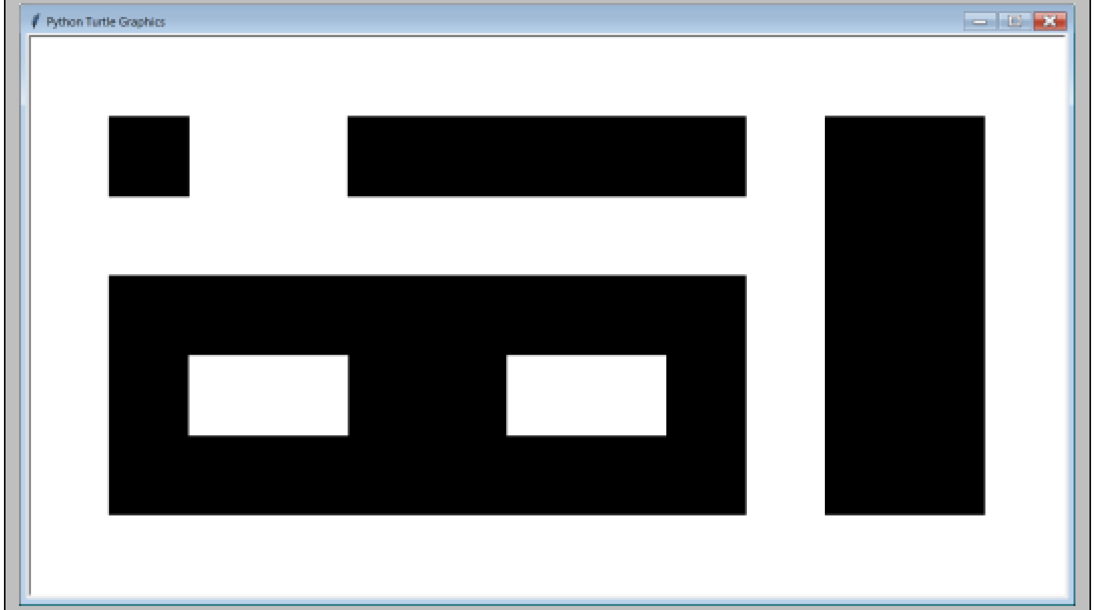
Do you see the two small rectangles inside the big one? Well, when all of these shapes are solid, filled in, rectangles, the two small, solid black rectangles are drawn on top of a bigger solid, filled in, rectangle which is also black. The result is the two smaller rectangles are not visible. The only way to make them visible is to change their color.

Changing Colors

When execute a graphics program, the background is white and the drawing color is black by default. In computer science, "default" refers to what you get when you do not specify something explicitly. The **setColor** command allows you to change the drawing color. Using this, we can fix the problem of the previous program. Program **GraphicsLibrary17.py**, in Figure 6.39, inserts a **setColor** command between the third and fourth solid rectangles. The first 3 solid rectangles are drawn in **black** by default. After that, the drawing color switches to **white**. Now the two small solid rectangles are drawn in **white** on top of the big **black** rectangle. That makes them visible. Remember, any time you want to draw a small object on top of a big object, you must draw the big object first, and then draw the small object second, in a different color, to make it visible. If you draw the small object first, the big object will completely cover it up.

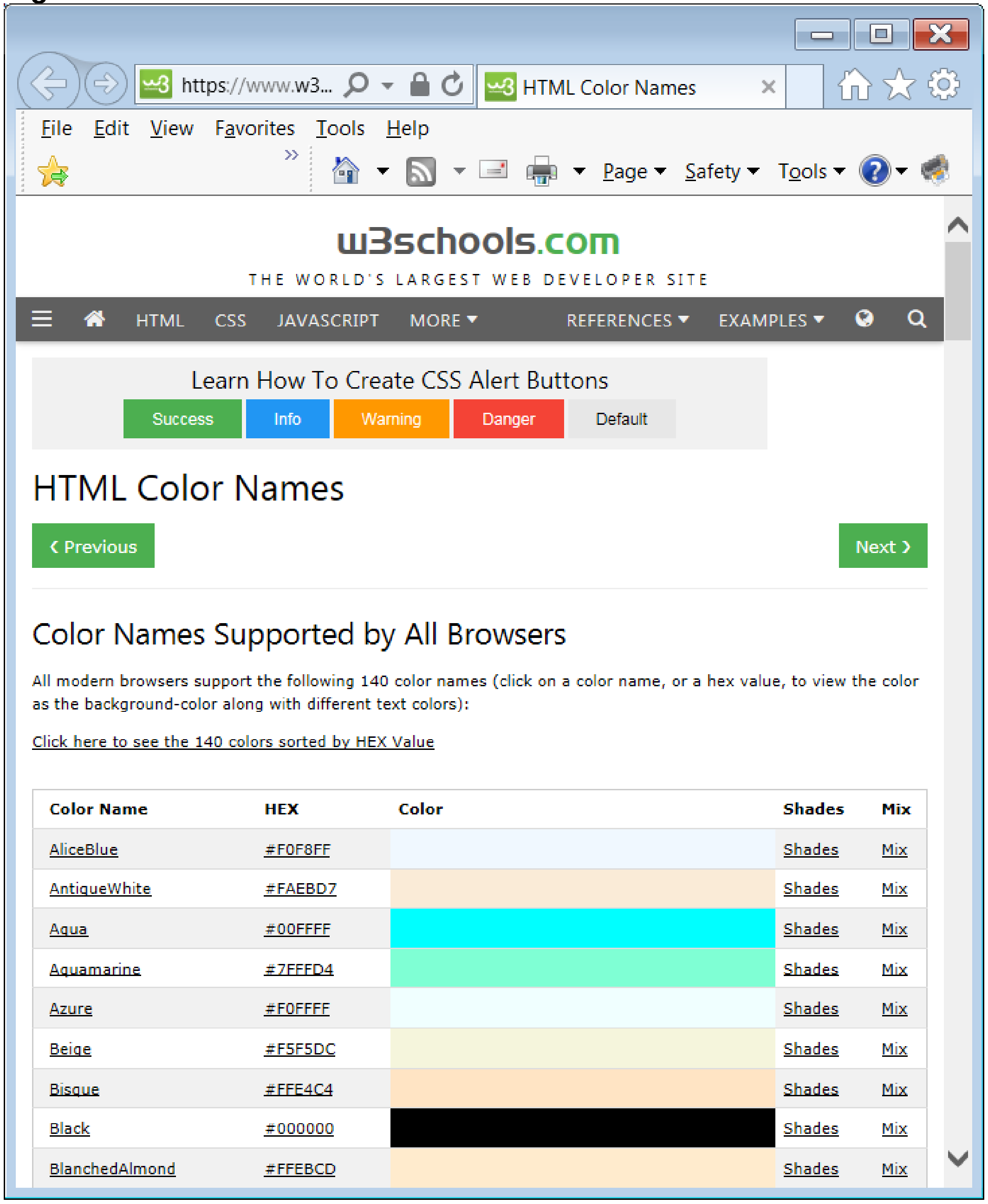
Figure 6.39

```
# GraphicsLibrary17.py
# This program demonstrates the <setColor>
# procedure of the <Graphics> library.
# Now the two missing rectangles from the
# previous program are visible.


from Graphics import *

beginGrfx(1300,700)

fillRectangle(100,100,200,200)
fillRectangle(400,100,900,200)
fillRectangle(100,300,900,600)
fillRectangle(1000,100,1200,600)
setColor("white")
fillRectangle(200,400,400,500)
fillRectangle(600,400,800,500)

endGrfx()

```



It should be noted that the **setColor** procedure of the **Graphics** library is very similar to the **color** procedure of the **turtle** library. Both can select from the same list of 140 different colors. The difference is that **setColor** has additional features that will be explained in Chapter VIII. Figure 6.40 shows the following website: **https://www.w3schools.com/colors/colors_names.asp** which provides a list of all of the "140 different colors" that I keep mentioning.

Figure 6.40



https://www.w3...
HTML Color Names
File Edit View Favorites Tools Help
Page Safety Tools
w3schools.com
THE WORLD'S LARGEST WEB DEVELOPER SITE
HTML CSS JAVASCRIPT MORE REFERENCES EXAMPLES
Success Info Warning Danger Default
❮ Previous
Next ❯

Learn How To Create CSS Alert Buttons

HTML Color Names

Color Names Supported by All Browsers

All modern browsers support the following 140 color names (click on a color name, or a hex value, to view the color as the background-color along with different text colors):

Click here to see the 140 colors sorted by HEX Value

Color Name	HEX	Color	Shades	Mix
AliceBlue	#F0F8FF	⬜	Shades	Mix
AntiqueWhite	#FAEBD7	⬜	Shades	Mix
Aqua	#00FFFF	🟦	Shades	Mix
Aquamarine	#7FFFD4	🟩	Shades	Mix
Azure	#F0FFFF	⬜	Shades	Mix
Beige	#F5F5DC	⬜	Shades	Mix
Bisque	#FFE4C4	🟧	Shades	Mix
Black	#000000	⬛	Shades	Mix
BlanchedAlmond	#FFEBCD	🟧	Shades	Mix

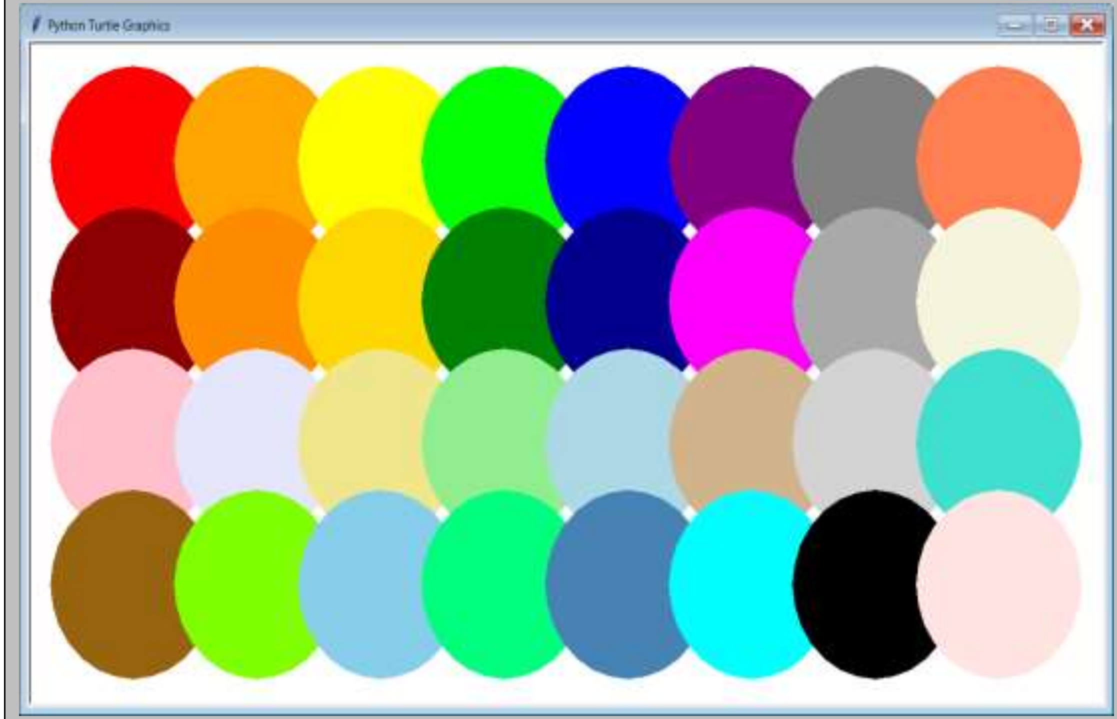
NOTE: While all 140 colors work in all "web browsers" and also work in Python on Windows based computers, some of these colors, like **Aqua**, do not work on a Mac.

Program **GraphicsLibrary18.py**, in Figure 6.41, demonstrates 32 of these 140 colors. It also demonstrates **fillCircle** which has the exact same arguments as **drawCircle**. There are a couple other things being demonstrated in this program. One is the fact that you can actually put 2 Python commands on the same line. In order to do this, you must put a semicolon (**;**) after the first command.

Figure 6.41

```
# GraphicsLibrary18.py
# This program demonstrates 32 of the colors
# that are available in Python.  For a complete
# list of all 140 colors, visit this website:
# https://www.w3schools.com/colors/colors_names.asp
# This program also demonstrates <fillCircle>.
# It also shows that you can put 2 Python
# commands on the same line if you end the
# first with a semicolon <;>


from Graphics import *

beginGrfx(1300,700)

radius = 100

setColor("red");          fillCircle(125,125,radius)
setColor("orange");       fillCircle(275,125,radius)
setColor("yellow");       fillCircle(425,125,radius)
setColor("green");        fillCircle(575,125,radius)
setColor("blue");         fillCircle(725,125,radius)
setColor("purple");       fillCircle(875,125,radius)
setColor("gray");         fillCircle(1025,125,radius)
setColor("coral");        fillCircle(1175,125,radius)

setColor("dark red");     fillCircle(125,275,radius)
setColor("dark orange");  fillCircle(275,275,radius)
setColor("gold");         fillCircle(425,275,radius)
setColor("dark green");   fillCircle(575,275,radius)
setColor("dark blue");    fillCircle(725,275,radius)
setColor("magenta");      fillCircle(875,275,radius)
setColor("dark gray");    fillCircle(1025,275,radius)
setColor("beige");        fillCircle(1175,275,radius)

setColor("pink");         fillCircle(125,425,radius)
setColor("lavender");     fillCircle(275,425,radius)
setColor("khaki");        fillCircle(425,425,radius)
setColor("light green");  fillCircle(575,425,radius)
setColor("light blue");   fillCircle(725,425,radius)
setColor("tan");          fillCircle(875,425,radius)
setColor("light gray");   fillCircle(1025,425,radius)
setColor("turquoise");    fillCircle(1175,425,radius)

setColor("brown");        fillCircle(125,575,radius)
setColor("chartreuse");   fillCircle(275,575,radius)
setColor("sky blue");     fillCircle(425,575,radius)
setColor("spring green"); fillCircle(575,575,radius)
setColor("steel blue");   fillCircle(725,575,radius)
setColor("cyan");         fillCircle(875,575,radius)
setColor("black");        fillCircle(1025,575,radius)
setColor("misty rose");   fillCircle(1175,575,radius)

endGrfx()

```

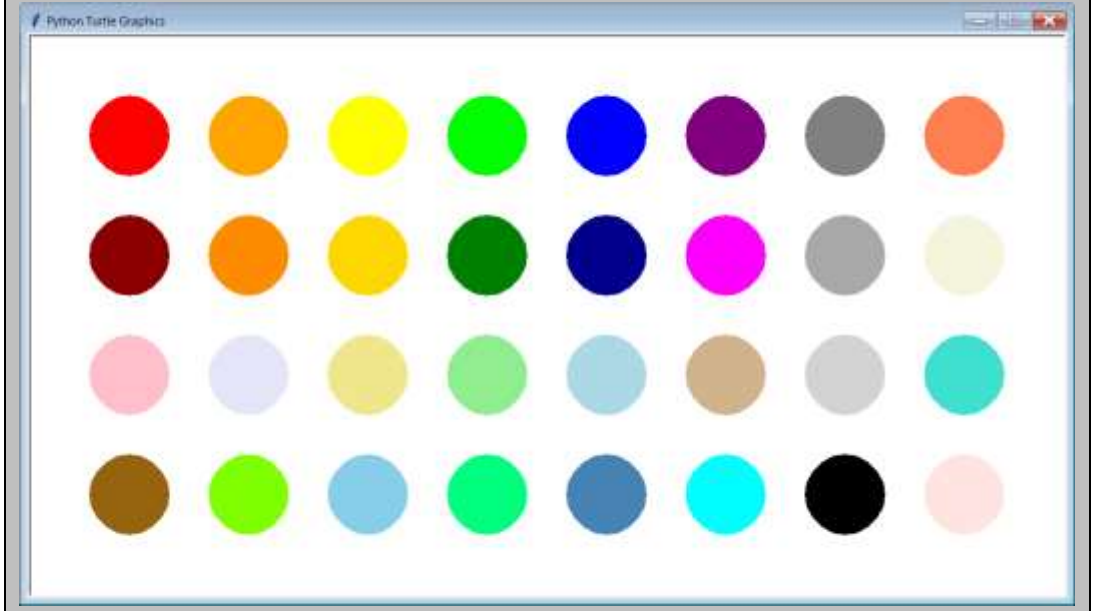
radius = 100



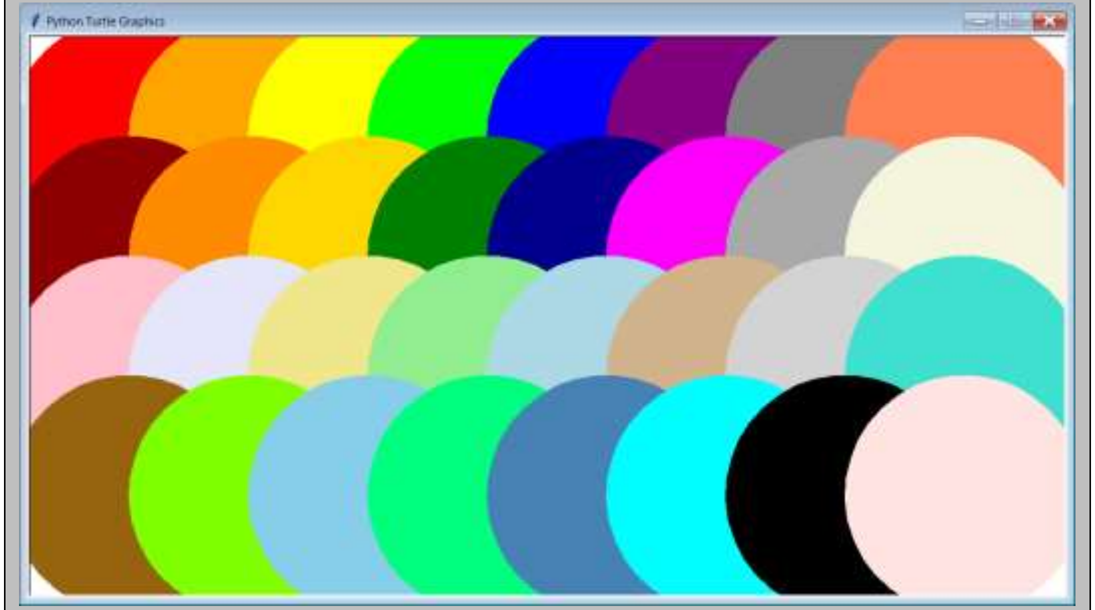
Notice that a variable is used for the **radius** argument. By doing this, I can change the radius of all of the circles simply by changing one single number. Figure 6.42 shows what the outputs would be if I changed the **radius** variable to **50** and **150**.

Figure 6.42

radius = 50



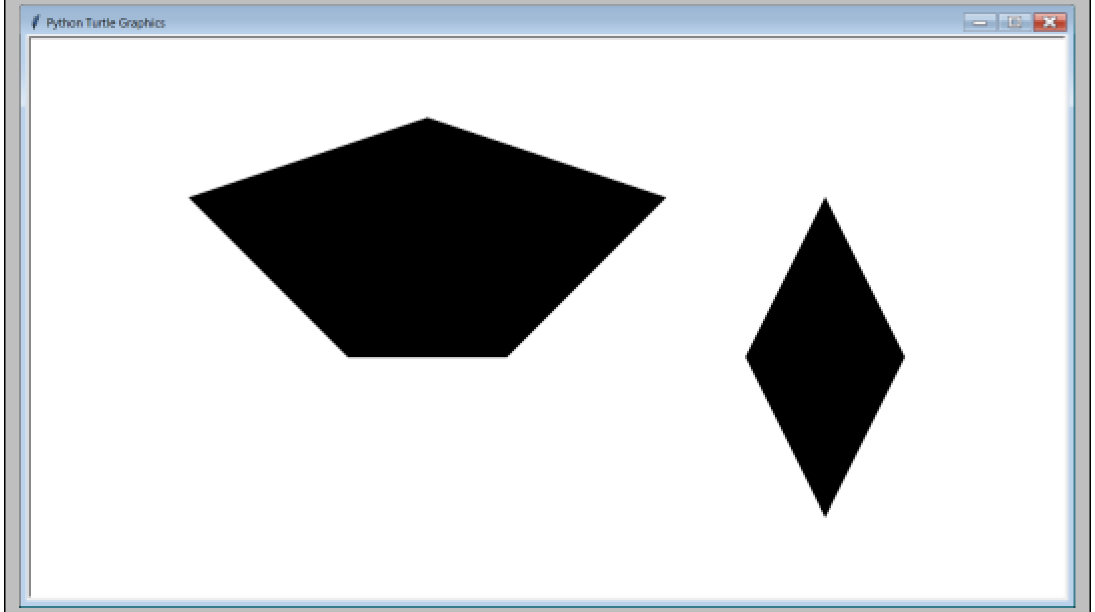
radius = 150



Program **GraphicsLibrary19.py**, in Figure 6.43, demonstrates **fillPolygon** which uses the exact same arguments as **drawPolygon**.

Figure 6.43

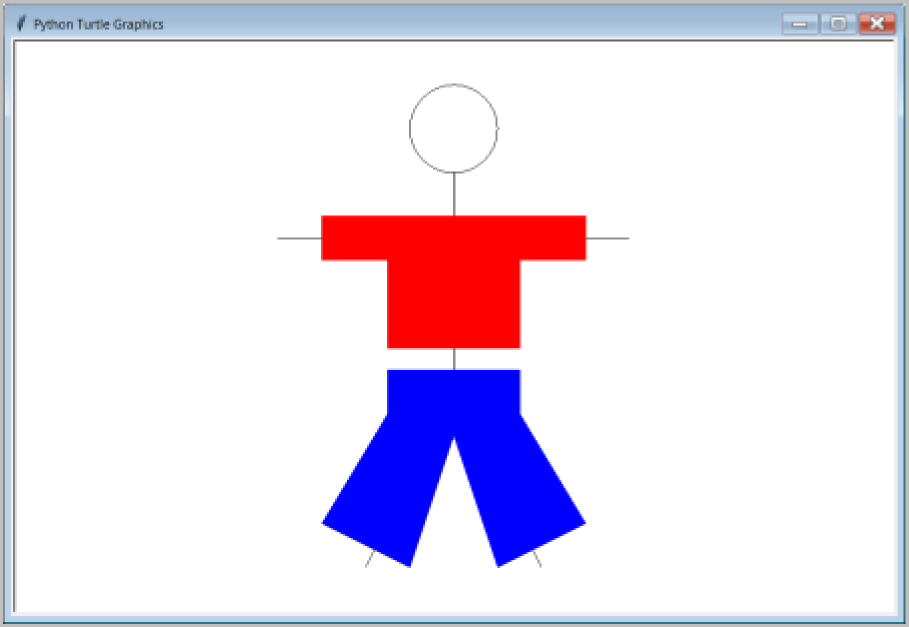
```python
# GraphicsLibrary19.py
# This program demonstrates the <fillPolygon>
# procedure of the <Graphics> library.
# The arguments for <fillPolygon> are
# exactly the same as for <drawPolygon>.


from Graphics import *

beginGrfx(1300,700)

fillPolygon([500,100,800,200,600,400,400,400,200,200])
fillPolygon([900,400,1000,200,1100,400,1000,600])

endGrfx()

```



Program **GraphicsLibrary20.py**, in Figure 6.44, demonstrates that the different polygons in your program can be filled with different colors.

Figure 6.44

```
# GraphicsLibrary20.py
# This program demonstrates that the different polygons
# in your program can be filled with different colors.


from Graphics import *

beginGrfx(1000,650)

drawCircle(500,100,50)
drawLine(500,150,500,400)
drawLine(500,400,400,600)
drawLine(500,400,600,600)
drawLine(300,225,700,225)
setColor("blue")
fillPolygon([425,375,425,425,350,550,450,600,
500,450,550,600,650,550,575,425,575,375])
setColor("red")
fillPolygon([350,200,650,200,650,250,575,250,
575,350,425,350,425,250,350,250])

endGrfx()
```

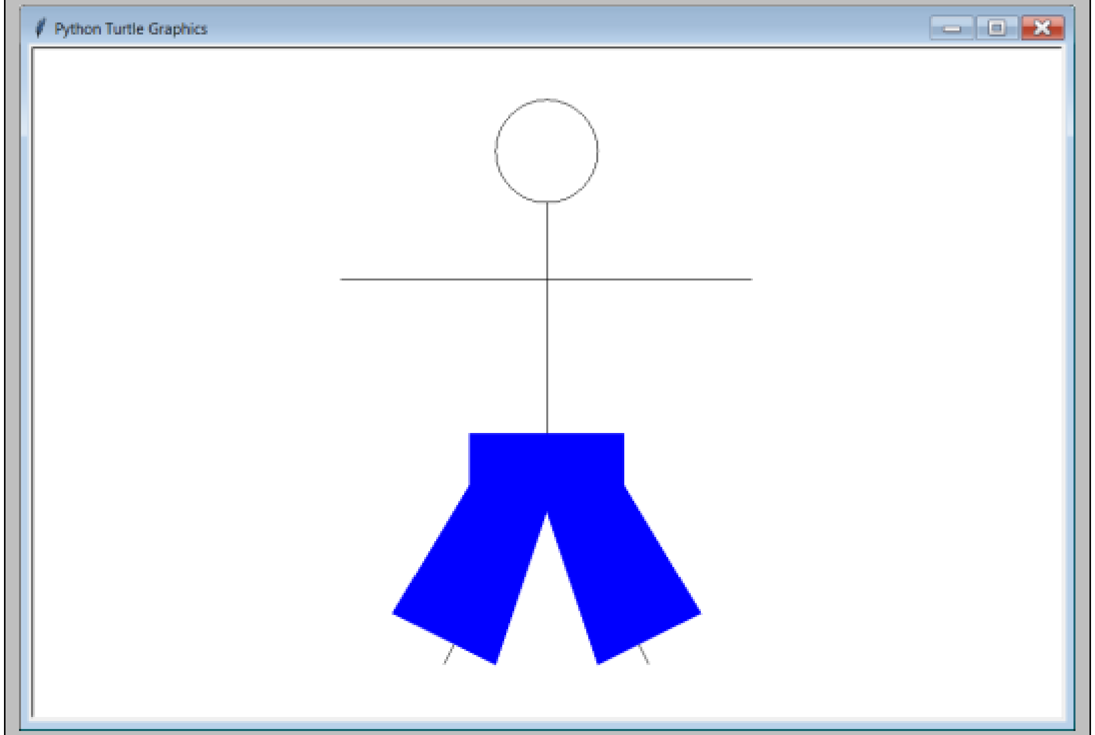


We have seen that the arguments for procedures **drawPolygon** and **fillPolygon** are "flexible". This does not mean that you can just use any number of arguments. There are a couple restrictions. First, since every X value must have a corresponding Y value, the number of integer arguments must be even. Second, since a polygon must have at least 3 sides, the number of integer arguments must be at least 6.

Program **GraphicsLibrary21.py**, in Figure 6.45, is essentially a copy of program **GraphicsLibrary20.py**. The one difference is that the last integer argument has been removed from the second **fillPolygon** command. This means the last X value does not have a corresponding Y value. The program will execute and display most of the output until it reaches this command. Then the output window closes and there is a run-time error message in the **Run I/O** window.

Figure 6.45

```
# GraphicsLibrary21.py
# This program demonstrates what happens when
# <drawPolygon> or <fillPolygon> is called with
# an incorrect number of integer arguments.
# Since the integer arguments represent
# coordinate points, the number of integer arguments
# must always be even.  Since a polygon must have
# at least 3 sides, the number of integer arguments
# must be at least 6.  If either condition is not
# met, a special error message is displayed.


from Graphics import *

beginGrfx(1000,650)

drawCircle(500,100,50)
drawLine(500,150,500,400)
drawLine(500,400,400,600)
drawLine(500,400,600,600)
drawLine(300,225,700,225)
setColor("blue")
fillPolygon([425,375,425,425,350,550,450,600,500,450,550,
600,650,550,575,425,575,375])
setColor("red")
fillPolygon([350,200,650,200,650,250,575,250,
575,350,425,350,425,250,350])

endGrfx()
```



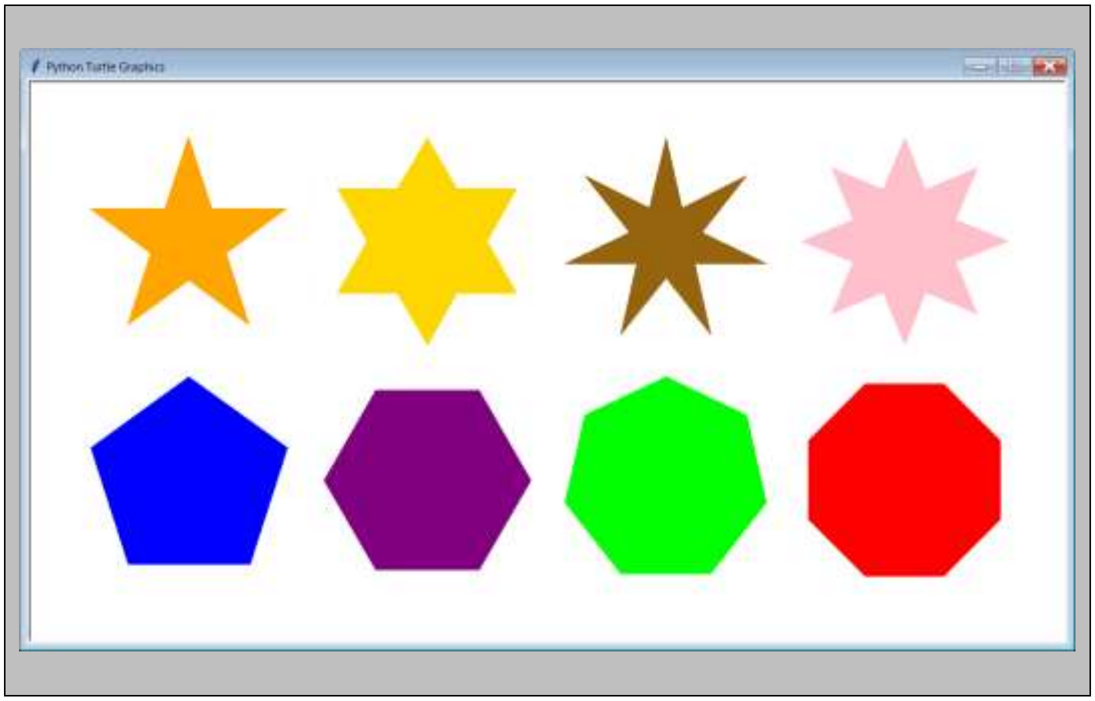
After 1 second, the graphics window closes and this is displayed in the **Run I/O** window:

```
 ----jGRASP exec: python GraphicsLibrary21.py
Polygon Error in line 25
When using drawPolygon/fillPolygon
you must have an even # of int arguments.

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

Program **GraphicsLibrary22.py**, in Figure 6.46, demonstrates **fillStar** and **fillRegularPolygon**. As with the other "fill" procedures, both of these have the exact same arguments as their "draw" counterparts.

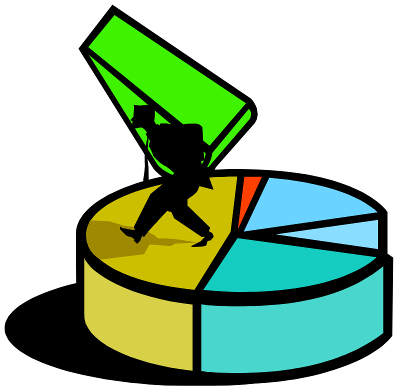
Figure 6.46

```
# GraphicsLibrary22.py
# This program demonstrates the <fillRegularPolygon>
# and <fillStar> procedures.  Both of these have the
# same exact arguments as their "draw" counterparts.


from Graphics import *

beginGrfx(1300,700)

setColor("orange")
fillStar(200,200,130,5)
setColor("gold")
fillStar(500,200,130,6)
setColor("brown")
fillStar(800,200,130,7)
setColor("pink")
fillStar(1100,200,130,8)
setColor("blue")
fillRegularPolygon(200,500,130,5)
setColor("purple")
fillRegularPolygon(500,500,130,6)
setColor("green")
fillRegularPolygon(800,500,130,7)
setColor("red")
fillRegularPolygon(1100,500,130,8)

endGrfx()

```



Drawing Solid Ovals, Pie Wedges and PAC-MAN

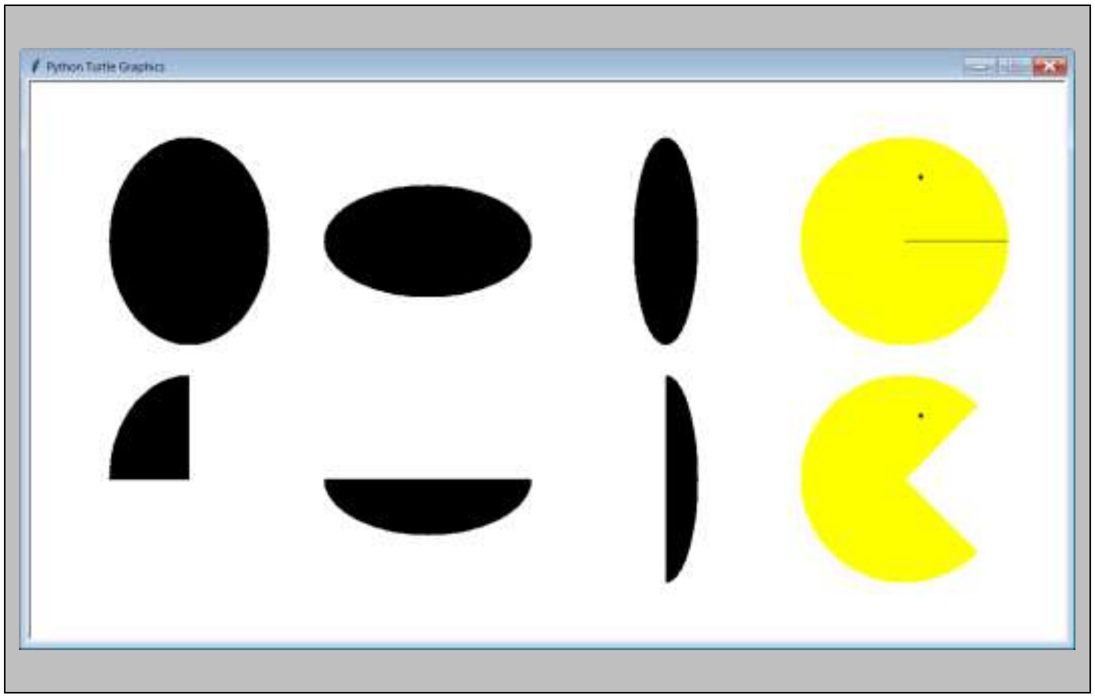
The next program demonstrates **fillOval** and **fillArc**. Now a **fillOval** procedure might have been expected, but **fillArc** may seem a little strange. Arcs are not enclosed like rectangles, circles, and ovals. Well, remember that an arc is a *piece* of an oval. In the same way, a *filled arc* is a "piece" of a *filled oval*. Imagine a piece of pie. The pie is the solid circle. The piece looks like a wedge. What is left over looks something like PAC-MAN.



The arguments for **fillOval** are the same as **drawOval**. The arguments for **fillArc** are also the same as **drawArc**. Program **GraphicsLibrary23.py**, in Figure 6.47, will draw 4 solid ovals. Beneath each solid oval, will be a partial oval, or "filled arc", with the same values for both radii.

Figure 6.47

```
# GraphicsLibrary23.py
# This program demonstrates <fillOval> & <fillArc>.
# In the same way that an arc is a piece of an oval
# a "filled arc" is a piece of a "filled oval".


from Graphics import *

beginGrfx(1300,700)

fillOval(200,200,100,130)
fillOval(500,200,130,70)
fillOval(800,200,40,130)
setColor("yellow")
fillOval(1100,200,130,130)
setColor("black")
drawLine(1100,200,1230,200)
drawPoint(1120,120)
fillArc(200,500,100,130,270,360)
fillArc(500,500,130,70,90,270)
fillArc(800,500,40,130,0,180)
setColor("yellow")
fillArc(1100,500,130,130,135,45)
setColor("black")
drawPoint(1120,420)

endGrfx()

```



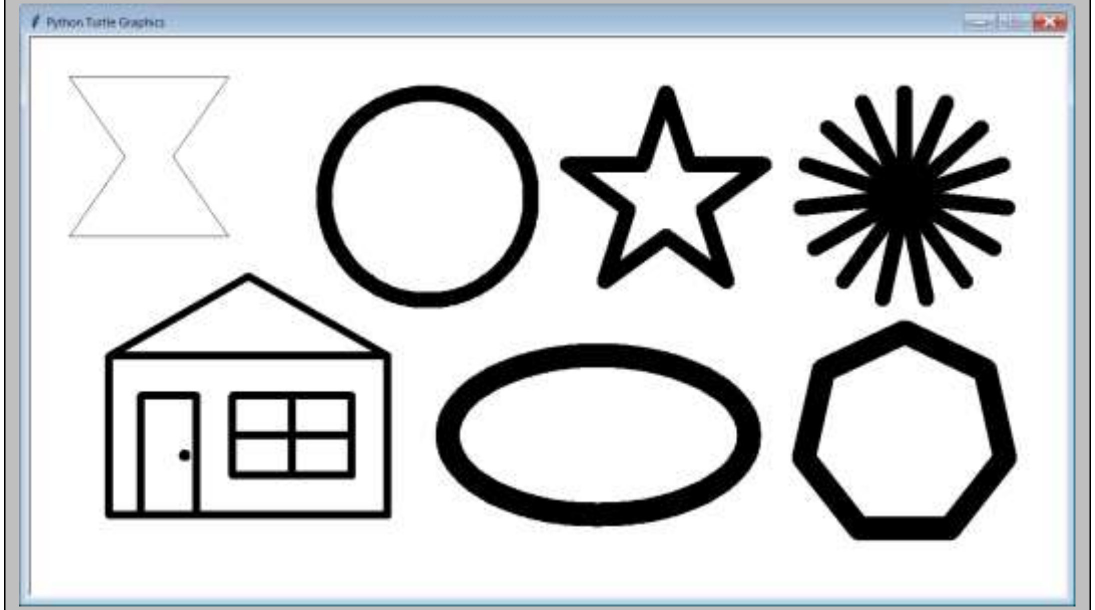
Changing the "width"

You may remember that the **turtle** library has a **width** procedure that can make the lines thicker. We can do the same this with the **Graphics** library. Program **GraphicsLibrary24.py**, in Figure 6.48, draws several different images using the exact same process as before; however, the lines are made thicker because the **width** procedure is used on multiple occasions.

Figure 6.48

```
# GraphicsLibrary24.py
# This program demonstrates that the <width>
# command from "Turtle Graphics" can also be
# used with commands from the <Graphics> library.


from Graphics import *

beginGrfx(1300,700)

width(10)
drawRectangle(100,400,450,600)
```

```
drawLine(100,400,275,300)
drawLine(450,400,275,300)
drawRectangle(255,450,405,550)
drawLine(330,450,330,550)
drawLine(255,500,405,500)
drawRectangle(140,450,210,600)
drawPoint(195,525)

width(20)
drawCircle(500,200,130)
drawStar(800,200,130,5)
drawBurst(1100,200,130,15)

width(30)
drawOval(715,500,190,100)
drawRegularPolygon(1100,500,130,7)

width(1) # back to default
drawPolygon([50,50,250,50,180,150,250,250,50,250,120,150])

endGrfx()
```



NOTE: The reason that we are able to use **width** with the **Graphics** library is actually very simple. The **Graphics** library imports the **turtle** library. That is why your graphics window always says "**Python Turtle Graphics**" at the top even though we are not using *Turtle Graphics*.

6.8 Graphics Library Reference Information

You might wonder how you can remember all the procedures of the **Graphics** library. Well, you are not expected to. Using reference materials in technological fields is quite normal. I have created a document called **Graphics Library Reference.docx** (and **.pdf**) which gives you detailed information on all of the **Graphics** library procedures. Figure 6.49 shows the first page of this file.

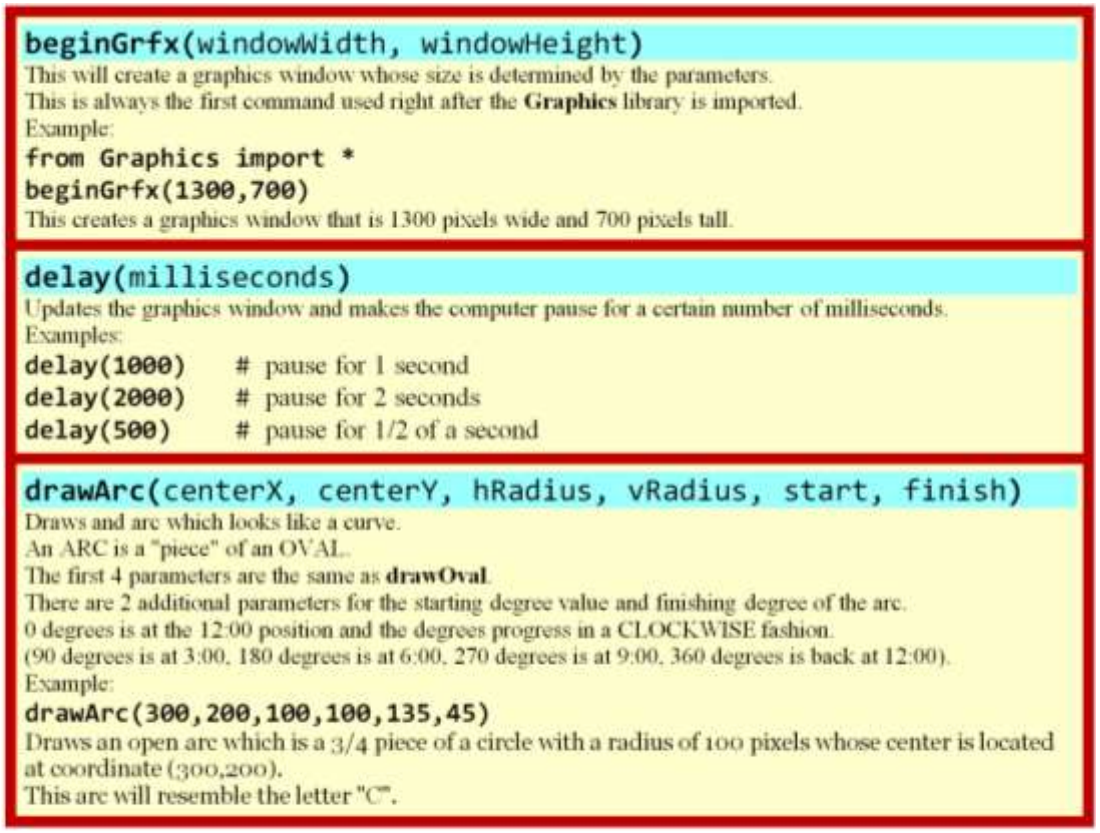
Figure 6.49

Mr. Schram's Graphics Library for Python

This version of the **Graphics** library was written by Mr. John Schram 10/3/18 for use in first year Computer Science 1 or Computer Science 1-Honors / PreAPCS.

While built on "Turtle Graphics", the procedures and functions below allow graphics programming with more traditional graphics commands for greater convenience. It was inspired by a similar graphics library created by Mr. Leon Schram and is designed to operate in a manner similar to that of the **Expo** class that we created for "Exposure Java".

This code is free software. You can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation. This code is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.



beginGrfx(windowWidth, windowHeight)

This will create a graphics window whose size is determined by the parameters.
This is always the first command used right after the **Graphics** library is imported.
Example:

```
from Graphics import *
beginGrfx(1300,700)
```

This creates a graphics window that is 1300 pixels wide and 700 pixels tall.

delay(milliseconds)

Updates the graphics window and makes the computer pause for a certain number of milliseconds.
Examples:

```
delay(1000)    # pause for 1 second
delay(2000)    # pause for 2 seconds
delay(500)     # pause for 1/2 of a second
```

drawArc(centerX, centerY, hRadius, vRadius, start, finish)

Draws and arc which looks like a curve.
An ARC is a "piece" of an OVAL.
The first 4 parameters are the same as **drawOval**.
There are 2 additional parameters for the starting degree value and finishing degree of the arc.
0 degrees is at the 12:00 position and the degrees progress in a CLOCKWISE fashion.
(90 degrees is at 3:00, 180 degrees is at 6:00, 270 degrees is at 9:00, 360 degrees is back at 12:00).
Example:

```
drawArc(300,200,100,100,135,45)
```

Draws an open arc which is a 3/4 piece of a circle with a radius of 100 pixels whose center is located at coordinate (300,200).
This arc will resemble the letter "C".

Things to Remember about the Graphics Library

- **The Graphics library is not part of Python.**
- **This library was created to allow graphics programming with more traditional, *coordinate-based* graphics commands for greater convenience.**
- **In order to use the Graphics library, the file Graphics.py must be in the same folder/directory as the Python file that calls the Graphics library subroutines.**
- **Students will NOT be required to memorize the functions and procedures of the Graphics library. They will instead be provided with Graphics Library Reference documentation to use during labs and tests.**

6.9 Displaying Graphics Text

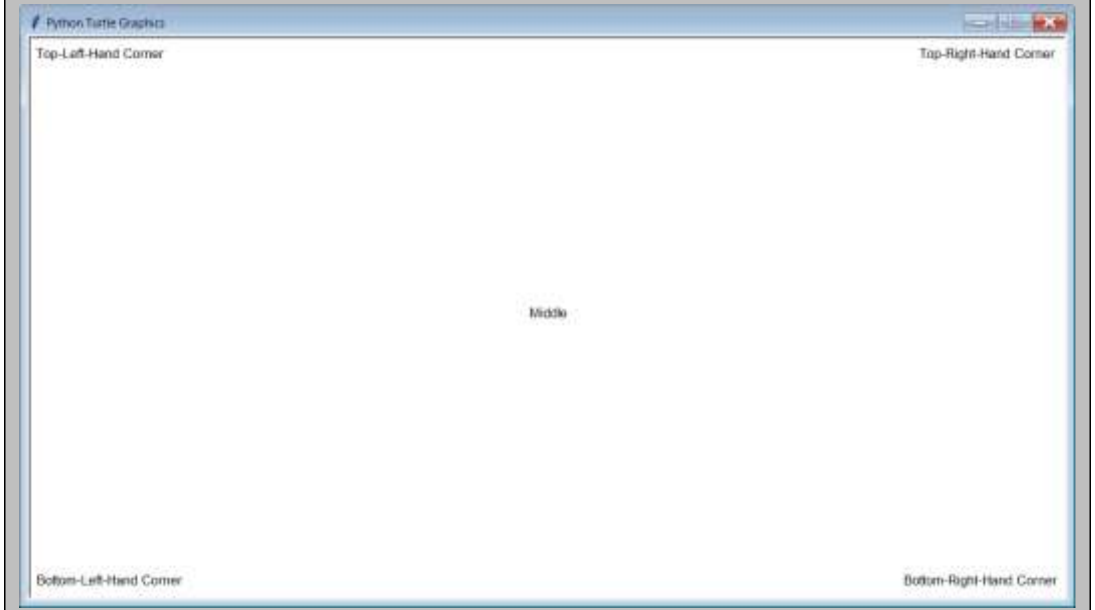
We have been displaying text in a “text window” since Chapter III. Now, we are going to display text on the graphics screen. The **print** command is great, but it only works in a text window. If you want to display text in a graphics window, you can use the **drawString** procedure of the **Graphics** library.

drawString can use up to 6 arguments. The first 3 are required. The other 3 are optional. The first argument is the “string” that you wish to “draw.” Remember that *string literals* go inside quotes. After that there are 2 integer arguments. These represent the X and Y coordinate values of the beginning of the string.

Program **GraphicsLibrary25.py**, in figure 6.50, displays 5 strings. For now, we are just focusing on the required arguments. We will look at the optional arguments in the next program example.

Figure 6.50

```
# GraphicsLibrary25.py
# This program demonstrates the <drawString>
# procedure of the <Graphics> library.  With
# <drawString("Hello World",x,y)>, the string
# "Hello World" will be displayed starting at
# coordinate (x,y).


from Graphics import *

beginGrfx(1300,700)

drawString("Top-Left-Hand Corner",10,30)
drawString("Top-Right-Hand Corner",1120,30)
drawString("Bottom-Left-Hand Corner",10,690)
drawString("Bottom-Right-Hand Corner",1100,690)
drawString("Middle",630,355)

endGrfx()

```



```
Top-Left-Hand Corner                                    Top-Right-Hand Corner

                              Middle

Bottom-Left-Hand Corner                                 Bottom-Right-Hand Corner
```

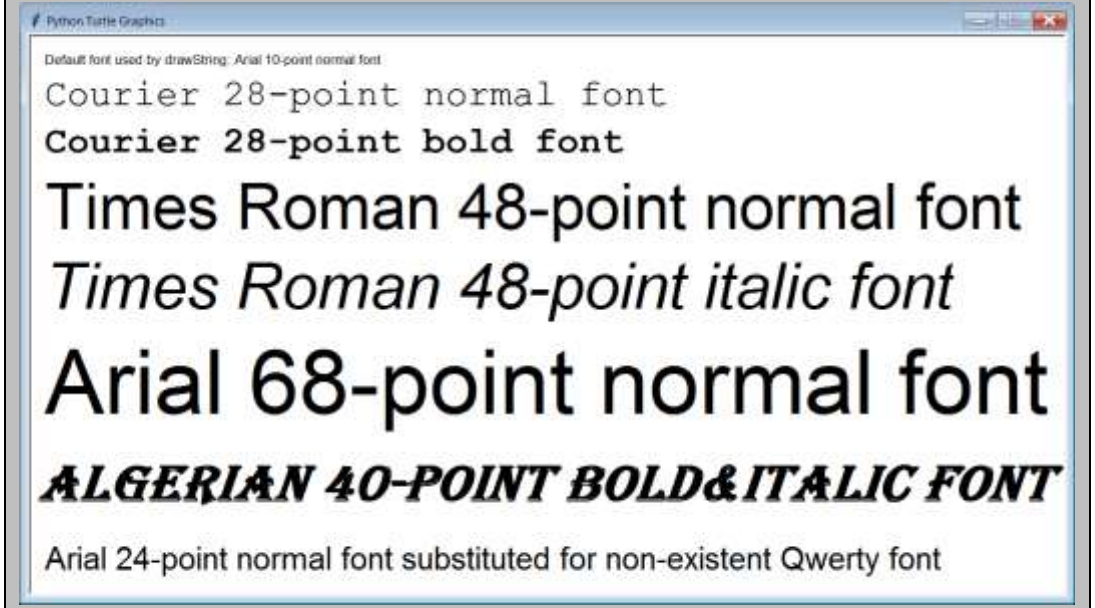
Displaying Different Graphics Fonts

When you look at the output of the previous program, you might be thinking, “OK, but that text is tiny.” Suppose you want to display bigger text, or **bold** text, or *italicized* text, or text in a different FONT, or any ***combination***. This is possible by using the 3 “optional arguments” for **drawString** that I mentioned earlier. By default, you get Arial, 10 point, normal (neither **bold**, nor *italic*) text. However, you can specify the font face, point size and style in the optional 4th, 5th and 6th arguments of **drawString**. Program **GraphicsLibrary26.py**, in figure 6.51, has several examples of **drawString** command which use the optional arguments to change the appearance of the text.

The first **drawstring** command does not use the optional arguments, so we get the default font for that one. Starting with the second **drawstring** we see the optional arguments being used. The 4th argument is the *font face* (name of the font). This is something like "Arial", "Courier", "ALGERIAN" or "Times Roman.” The 5th argument is the *font size*. This is simply an integer that controls the “point size” of the text. The bigger this integer – the bigger the text. The 6th and final argument is the *font style*. This can be one of 4 possible string literals. You either use "normal", **"bold”**, *"italic"* or ***"bold&italic"***. The last **drawString** command has an issue. It is using a *font face*, **"Qwerty"**, that does not exist. Notice that there is no error message. If you misspell the name of a font, or use a font that simply does not exist on that particular computer, the computer will simply substitute its default font of "Arial".

Figure 6.51

```
# GraphicsLibrary26.py
# This program demonstrates that the <drawString>
# procedure can have up to a total of 6 arguments.
# The optional 4th, 5th and 6th arguments specify
# the "Font Face" (name of the font), the "Font Size"
# and the "Font Style" which can be <"normal">,
# <"bold">, <"italic"> or <"bold&italic">.


from Graphics import *

beginGrfx(1300,700)

drawString("Default font used by drawString: Arial 10-point normal font",20,40)
drawString("Courier 28-point normal font",20,100,
"Courier",28,"normal")
drawString("Courier 28-point bold font",20,160,
"Courier",28,"bold")
drawString("Times Roman 48-point normal font",20,
260,"TimesRoman",48) # "normal" is default
drawString("Times Roman 48-point italic font",20,
360,"TimesRoman",48,"italic")
drawString("Arial 68-point normal font",20,500,
"Arial",68)
drawString("Algerian 40-point bold&italic font ",10,
600,"Algerian",40,"bold&italic")
drawString("Arial 24-point normal font substituted
for non-existent Qwerty font",20,680,"Qwerty",24)

endGrfx()

```



Default font used by drawString: Arial 10-point normal font

Courier 28-point normal font

Courier 28-point bold font

Times Roman 48-point normal font

Times Roman 48-point italic font

Arial 68-point normal font

ALGERIAN 40-POINT BOLD&ITALIC FONT

Arial 24-point normal font substituted for non-existent Qwerty font

Displaying Multiple Pieces of Information with drawString

Program **GraphicsLibrary27.py** in figure 6.52, demonstrates an important difference between **print** and **drawString**. Look at lines 17 and 18. You see the definition of variables **firstName** and **lastName**. If I wanted to display both, on right after the other, on the text screen, I would simply use **print(firstName, lastName)**. I could even add a "Hello" at the beginning with **print("Hello", firstName, lastName)**. So now I am going to try to do the same basic thing with **drawString**. When you try to execute the program, you get a syntax error almost instantly. The error message specifies that the **drawString** procedure on line 23 can have between 3 and 6 arguments, but we are using 8. The problem is that **"Hello ", firstName, lastName** actually counts as 3 separate arguments instead of just 1.

Figure 6.52

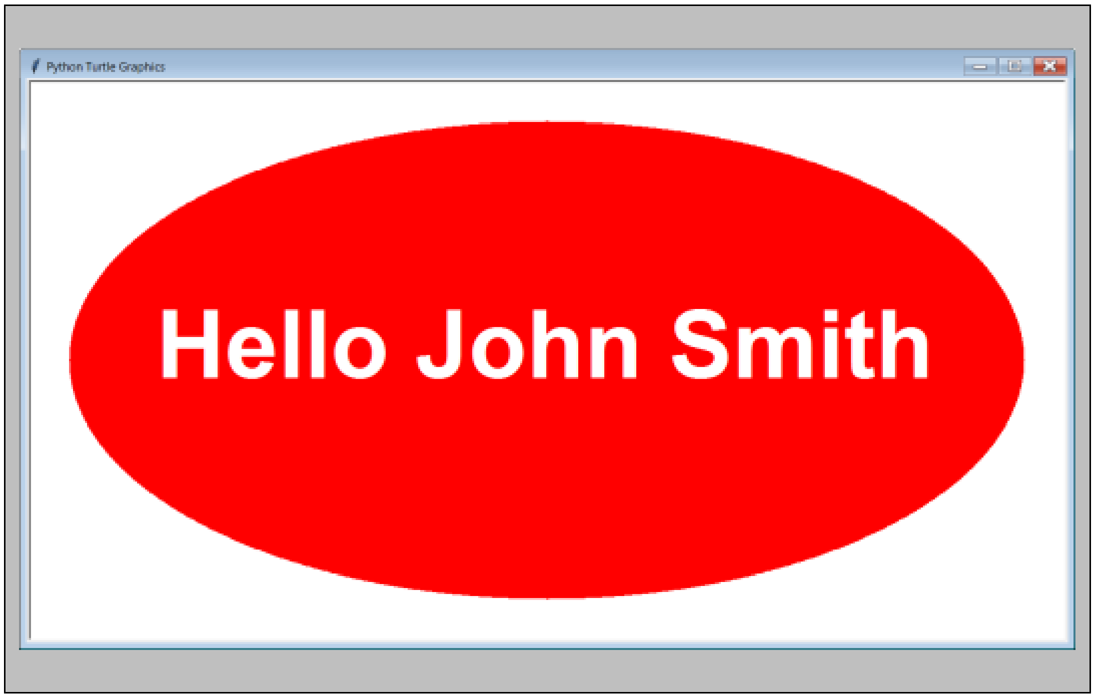
```
 1 # GraphicsLibrary27.py
 2 # This program demonstrates what happens when you try to
 3 # display multiple pieces of information with <drawString>
 4 # separated with commas like you do with <print>.
 5 # This does not work because the each separate piece of
 6 # information is its own argument.
 7
 8
 9 from Graphics import *
10
11 beginGrfx(1300,700)
12
13 setColor("red")
14 fillOval(650,350,600,300)
15 setColor("white")
16
17 firstName = "John "
18 lastName = "Smith"
19 drawString("Hello ",firstName,lastName,
160,400,"Arial",72,"bold")
20
21 endGrfx()
22
```

```
  ----jGRASP exec: python GraphicsLibrary27.py
 Traceback (most recent call last):
   File "GraphicsLibrary27.py", line 19, in <module>
     drawString("Hello",firstName,lastName,160,400,
"Arial",72,"bold")
 TypeError: drawString() takes from 3 to 6 positional
arguments but 8 were given

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

Program **GraphicsLibrary28.py**, in figure 6.53, fixes the issue of the previous program by using *string concatenation* to combine the 3 string values: "**Hello** ", **firstName** and **lastName** into a single argument.

Figure 6.53

```python
# GraphicsLibrary28.py
# This program demonstrates the proper way to display
# multiple pieces of information with <drawString>.
# The secret is to use String Concatenation to combine
# the different pieces of information into a single
# string argument.


from Graphics import *

beginGrfx(1300,700)

setColor("red")
fillOval(650,350,600,300)
setColor("white")

firstName = "John "
lastName = "Smith"
drawString("Hello "+firstName+lastName,160,400,
"Arial",72,"bold")

endGrfx()
```



Program **GraphicsLibrary29.py**, in figure 6.54, tries to use the same *string concatenation* technique from the previous program. This time there is a problem. The program computes the average of 4 numbers and them tries to display it with **drawString**. If the output were being displayed on the text screen, there would be no problem using **print("The average is",average)**. We know that if we begin our **drawString** command with **drawString("The average is",average,** we will get an error for having too many arguments. So instead, we are trying to combine **"The average is"** and **average** into a single argument. This time it does not work. Remember, we can add numbers. We can concatenate strings, but we cannot combine a number and a string.

Figure 6.54

```
# GraphicsLibrary29.py
# This program demonstrates that the concatenation
# trick does not work if one of the pieces of
# information is a number.


from Graphics import *

beginGrfx(1300,700)
```

```

setColor("red")
fillOval(650,350,600,300)
setColor("white")

average = (10 + 20 + 30 + 40) / 4
drawString("The average is "+average,105,400,
"Arial",72,"bold")

endGrfx()

```

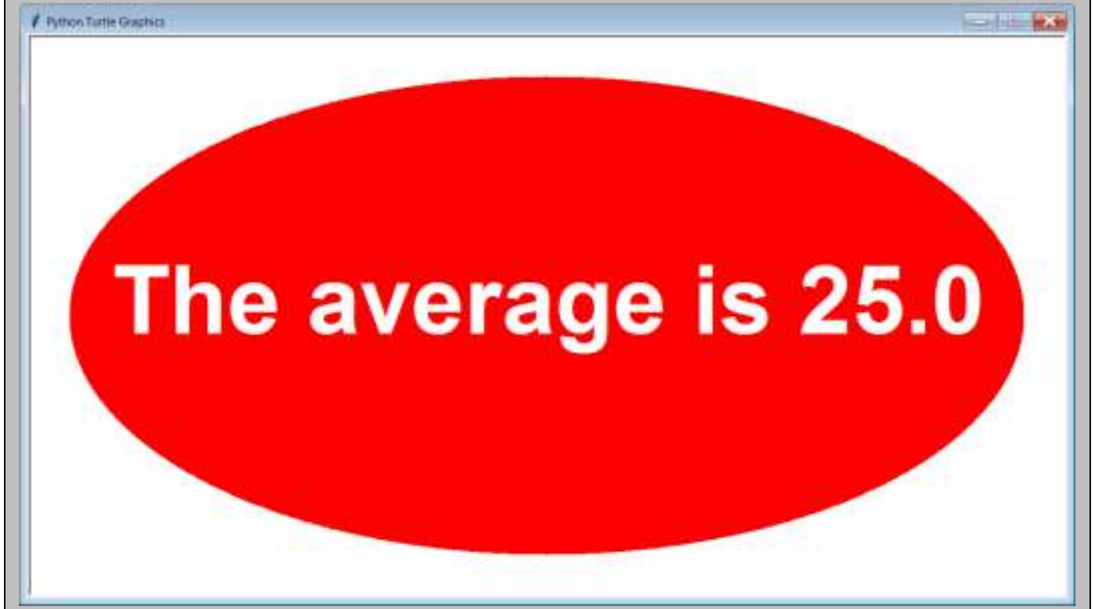
```
 ----jGRASP exec: python GraphicsLibrary29.py
Traceback (most recent call last):
  File "GraphicsLibrary29.py", line 16, in <module>
    drawString("The average is "+average,160,400,
"Arial",72,"bold")
TypeError: must be str, not float

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

So does that mean we just cannot display the value of numeric variables with **drawString**? Far from it. Program **GraphicsLibrary30.py**, in figure 6.55, fixes the problem of the previous program by using **str** to convert the real number variable **average** to a string value. Now there is no problem with the concatenation and things can be displayed properly.

Figure 6.55

```
# GraphicsLibrary30.py
# This program fixes the problem of the previous program
# by using <str> to convert the number to a string.
# Now it can be concatenated with other strings.


from Graphics import *
```

```

beginGrfx(1300,700)

setColor("red")
fillOval(650,350,600,300)
setColor("white")

average = (10 + 20 + 30 + 40) / 4
drawString("The average is "+str(average),
105,400,"Arial",72,"bold")

endGrfx()

```



The average is 25.0

6.10 Review: Functions vs. Procedures

Now that we have looked at many examples of both functions and procedures, I think we can get a better understanding of how they are similar and how they are different.

First, we will look at the **sqrt** function from the **math** library. This function executes the series of commands necessary to compute the *principal square root* of the argument. When this is done, the "square root" is returned so that we can use its value.

Second, we will look at the **drawCircle** procedure from the **Graphics** library. This procedure executes the series of commands necessary to draw the circle on the screen whose location and size are determines by the arguments. When this is done, the procedure is finished. There was no value computed. There is nothing that needs to be returned.

Notice that both the function and the procedure execute a series of commands to perform a desired task. In other words, both are *subroutines*. In the process of executing their series of commands, some subroutines compute a value that needs to be returned. These are the *functions*. In some languages they are called *return functions* or *return methods* but in Python we just call them "functions". There are other subroutines that contain a series of commands, but do not compute a value, and therefore do not return anything. These are the *procedures*. In some languages they are called *void functions* or *void methods* but in Python we just call them "procedures".

Subroutines, Functions and Procedures

A *subroutine* is a series of programming commands that performs a specific task.

A *function* is a subroutine that returns a value.

The subroutines in the **math** library are *functions*.

A *procedure* is a subroutine that does not return a value.

The subroutines in the **Graphics** library are mostly *procedures*.