Chapter IX

Modular Programming

Chapter IX Topics

9.1 Introduction

It is in the nature of people to create things that make our lives simpler and more enjoyable. To that end, it is also in the nature of people to create *tools* to make this process easier and more efficient.

The goal of a programming language is to make the creation of a program simpler. Working with the binary, base-2 numbers, better known as *machine code*, might make perfect sense to the computer, but for human being it is rather difficult and tedious. Some of the first programming *tools* created were translating programs like compilers and interpreters. This simplified the process of programming by allowing us to program (or "code" as many people now say) with English-like commands.

Another improvement with modern programming languages is the use of several intuitive *subroutines*. A subroutine is a group of programming commands combined into a single *module*. This single module performs a single task which is useful (like a *tool*). Once such a module is created, it is no longer necessary to use all the individual commands. Subroutines have different names in different programming languages. The language FORTRAN simply calls them *subroutines*. C++ calls them *void functions* and *return functions*. Java calls them *void methods* and *return methods*. Python, like Pascal, calls them *procedures* and *functions*.

Subroutines, Functions and Procedures Review

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.

In a way, it is like improving your English vocabulary. You could say, "I am so very tired that I cannot even move." You could also say, "I am exhausted." With increased vocabulary, you can communicate more effectively and more efficiently. The same is true for programming. The difference with programming is that it is not just a matter of learning more programming commands; it is also a matter of creating your own.

9.2 What is Modular Programming?

The title of this chapter is *Modular Programming* but what exactly is "Modular Programming?" It is possible to write a program with hundreds or even thousands of lines of code and not use any subroutines whatsoever. Such a program may work correctly, but making any corrections or will be very difficult.

Imagine the following program. Your program draws a very beautiful horse. This horse requires 2500 program statements. Now suppose that you have a gorgeous horse, but the tail is wrong. Fixing the tail problem is difficult with many program statements bunched together. On the other hand, if you had created several small modules for each part of the horse, making changes would be much simpler. If every group of related statements is placed in its own module (which would be a *procedure* or a *function* in Python) then you have taken an important step in designing your program.

Another terrific benefit is that subroutines have a name or *identifier*. Perhaps the tail requires fifty lines of programming code. Place each one of those statements into one procedure and call it **drawTail**. Anytime you need to draw the tail, you just say **drawTail**. You do not need to re-type the fifty lines of programming code. Also, any future programming that needs to fix or improve the tail is now simpler. Instead of searching through thousands of lines of programming code, you just look for the procedure called **drawTail**.

This business of combining common statements into their own modules is called *Modular Programming* with the philosophy: *One task, one module*.

Modular Programming

Modular programming is the process of placing statements that achieve a common purpose into its own module or subroutine.

An old programming saying says it well

One Task, One Module

9.3 Creating Simple Procedures

Creating programming "tools" in Python will start with a very simple program that displays a mailing address. There is nothing complicated about this program and there is nothing that demonstrates how to create subroutines. Program **SimpleProcedures01.py**, in Figure 9.1, shows a program that displays a mailing address and will be used to introduce the process of creating simple Python procedures.

Figure 9.1

```
 1 # SimpleProcedures01.py
 2 # This program displays a simple mailing address.
 3 # It will be used to demonstrate how to divide a
 4 # program into multiple user-created procedures.
 5
 6
 7 print()
 8 print("Kathy Smith")
 9 print("7003 Orleans Court")
10 print("Kensington, MD 20795")
11
```

```
 ----jGRASP exec: python SimpleProcedures01.py

Kathy Smith
7003 Orleans Court
Kensington, MD 20795

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

Up to this point, the only subroutines that have been use in your programs are those that are either part of the Python language or from my **Graphics** library. Essentially, you have only used subroutines that were created by other people. The main point of this chapter is learning how to create your own subroutines. Please realize that whether a library or subroutine is already provided by Python or created by you yourself, the process of calling subroutines is still the same.

Program **SimpleProcedures02.py**, in Figure 9.2, creates the exact same mailing address as the previous program. This time there are three procedures in the program. The previous address program statements have been placed in three separate procedures, called **displayName**, **displayStreetAddress** and **displayCityStateZip**. Each of these new procedures is *user-created*. Please observe that each procedure has a format similar to a control structure. First, there is a heading, following by a colon (:), then there is a set of indented programming commands.

In other languages like Java or C++, you have a special subroutine called the *main method* or the *main function*. This is not required in Python. The "main" section of the program is simply the group of programming commands at the bottom of the file that are not part of any subroutine. For the sake of readability, I will always place a comment here that says "**MAIN**" to clearly identify it. It is from this "**MAIN**" section that the most of the functions and procedures will be called.

Figure 9.2

```
1  # SimpleProcedures02.py
2  # This program introduces user-created procedures.
3  # The 3 program statements of the SimpleProcedures01.py
4  # program are now divided into 3 user-created procedures.
5  # Each procedure is simply called by using its name.
6
7
8  ############################
9  #  Procedure Definitions   #
10 ############################
11
12 def displayName():
13     print()
14     print("Kathy Smith")
15
16
17 def displayStreetAddress():
18     print("7003 Orleans Court")
19
20
21 def displayCityStateZip():
22     print("Kensington, MD 20795")
23
```

```
24
25
26 ############################
27 #  MAIN: Procedure Calls   #
28 ############################
29
30 displayName()
31 displayStreetAddress()
32 displayCityStateZip()
```

```
----jGRASP exec: python SimpleProcedures02.py

Kathy Smith
7003 Orleans Court
Kensington, MD 20795

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

User-Created Procedure Format

A user-defined procedure requires:

- The Python keyword **def**
- A heading, which includes the procedure name
- A set of parentheses ( )
- Just like conditional statements, procedure headings end with a colon **:** .
- An indented body of program statements beneath the heading

```
def example():
    print("This is an example of a")
    print("user-defined procedure.")
```

Sometimes students will ask why it is necessary to break up such a small program into different procedures. It does not seem to be necessary. After all, the first program worked just fine with no subroutines. Now, I could include examples of professional software, which absolutely would not work without subroutines; however, such code is thousands of lines long and more than a bit complex. Please realize the process of breaking a program up into different subroutines is the same whether the program has 3 program statements or 3000 program statements. Simplistic examples are often used to help the understanding of a concept – not to show examples of real world programs which would overwhelming in a first year computer science course.

Potential Procedure Problems

Program **SimpleProcedures03.py**, in Figure 9.3, demonstrates what happens when you do not indent the program statements that are part of a procedure. You will get the same error message that you get when you do not indent the program statements that are being controlled by a control structure. Remember, in other programming languages, indenting in recommended. In Python, it is required.

Figure 9.3

```
 1 # SimpleProcedures03.py
 2 # This program demonstrates that, as with
 3 # control structures, in Python indenting
 4 # subroutine commands is required.
 5
 6
 7 ############################
 8 #  Procedure Definitions  #
 9 ############################
10
11 def displayName():
12 print()
13 print("Kathy Smith")
14
15
16 def displayStreetAddress():
17 print("7003 Orleans Court")
18
```

```
19
20 def displayCityStateZip():
21 print("Kensington, MD 20795")
22
23
24
25 ############################
26 #  MAIN: Procedure Calls  #
27 ############################
28
29 displayName()
30 displayStreetAddress()
31 displayCityStateZip()
32
```

```
----jGRASP exec: python SimpleProcedures03.py
 File "SimpleProcedures03.py", line 12
   print()
       ^
IndentationError: expected an indented block

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

Updated Indentation Rule

In most languages, indenting the program statements that are "controlled" by control structures *or part of a subroutine* is recommended.

In Python, both are required.

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

Remember, it is not good enough simply to *indent* in Python, you must indent consistently. Program **SimpleProcedures04.py**, in Figure 9.4, demonstrates what happens when the indentation is not consistent. In the **displayName** procedure, the first program statement, on line 11, is indented 3 spaces. The second program statement, on line 12, is indented 4 spaces. Python is not happy and gives you an **IndentationError: unexpected indent**.

Figure 9.4

```
 1 # SimpleProcedures04.py
 2 # This program demonstrates what happens
 3 # when indentation is inconsistent.
 4
 5
 6 ############################
 7 #  Procedure Definitions   #
 8 ############################
 9
10 def displayName():
11     print()
12      print("Kathy Smith")
13
14
15 def displayStreetAddress():
16    print("7003 Orleans Court")
17
18
19 def displayCityStateZip():
20    print("Kensington, MD 20795")
21
22
23
24 ############################
25 #  MAIN: Procedure Calls   #
26 ############################
27
28 displayName()
29 displayStreetAddress()
30 displayCityStateZip()
```

```
----jGRASP exec: python SimpleProcedures04.py
 File "SimpleProcedures04.py", line 12
   print("Kathy Smith")
   ^
IndentationError: unexpected indent

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

Program **SimpleProcedures05.py**, in Figure 9.5, demonstrates an error that very similar to that of the previous program. What is different now is, you cannot see the error. This is a case of invisibly inconsistent indentation. My recent alliteration aside, how is this possible? The **displayName** procedure in this program looks just like the one in program **SimpleProcedures02.py**, which worked just fine. Here is the problem. In this program, the program statement on line 12 is indented with a <tab>, while the program statement on line 13 is indented with 3 spaces. While both look the same to us, they are very different to the computer.

Figure 9.5

```
 1 # SimpleProcedures05.py
 2 # "Invisibly Inconsistent Indentation Error"
 3 # You can indent with a <tab> or with some
 4 # spaces, but you need to be consistent.
 5
 6
 7 ############################
 8 #  Procedure Definitions  #
 9 ############################
10
11 def displayName():
12     print()               # Indented with a <tab>
13     print("Kathy Smith") # Indented with 3 spaces
14
15
```

```
16 def displayStreetAddress():
17     print("7003 Orleans Court")
18
19
20 def displayCityStateZip():
21     print("Kensington, MD 20795")
22
23
24
25 ############################
26 #  MAIN: Procedure Calls   #
27 ############################
28
29 displayName()
30 displayStreetAddress()
31 displayCityStateZip()
32
```

```
  ----jGRASP exec: python SimpleProcedures05.py
   File "SimpleProcedures05.py", line 13
     print("Kathy Smith") # Indented with 3 spaces
                                                  ^
  IndentationError: unindent does not match
any outer indentation level

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

So how can you tell tabs versus spaces? If you are using **jGRASP**, there is a way. Just click at the beginning of the line. If the cursor looks like it normally does, with a flashing symbol that resembles a capital L, as shown in Figure 9.6, then you have spaces.

Figure 9.6

```
11 def displayName():
12     print()                 # Indented with a <tab>
13     print("Kathy Smith") # Indented with 3 spaces
```

On the other hand, if the capital L looks much wider than it normally does, as shown in Figure 9.7, then you have a tab.

Figure 9.7

```
11 def displayName():
12 ⟶print()                  # Indented with a <tab>
13    print("Kathy Smith") # Indented with 3 spaces
```

We have one more program in this section. Program **SimpleProcedures06.py**, in Figure 9.8, demonstrates what happens when things are out of order. The proper order of things, at least in Python, is to first define your procedures (or functions) at the top of the program, and then you can call them at the bottom. If you reverse this and put the procedure calls before the definitions the computer will attempt to call procedures which have not been defined (yet), which is almost the exact error message that you get.

Figure 9.8

```
 1 # SimpleProcedures06.py
 2 # This program shows that in Python,
 3 # the "Procedure Definitions" must
 4 # come before the "Procedure Calls".
 5
 6
 7 ############################
 8 #  MAIN: Procedure Calls  #
 9 ############################
10
11 displayName()
12 displayStreetAddress()
13 displayCityStateZip()
14
15
16
17 ############################
18 #  Procedure Definitions  #
19 ############################
20
```

```
21 def displayName():
22     print()
23     print("Kathy Smith")
24
25
26 def displayStreetAddress():
27     print("7003 Orleans Court")
28
29
30 def displayCityStateZip():
31     print("Kensington, MD 20795")
32
```

```
 ----jGRASP exec: python SimpleProcedures06.py
Traceback (most recent call last):
File "SimpleProcedures06.py", line 11, in <module>
    displayName()
NameError: name 'displayName' is not defined

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

9.4 Graphics Programs with Procedures

Turtle Graphics Examples

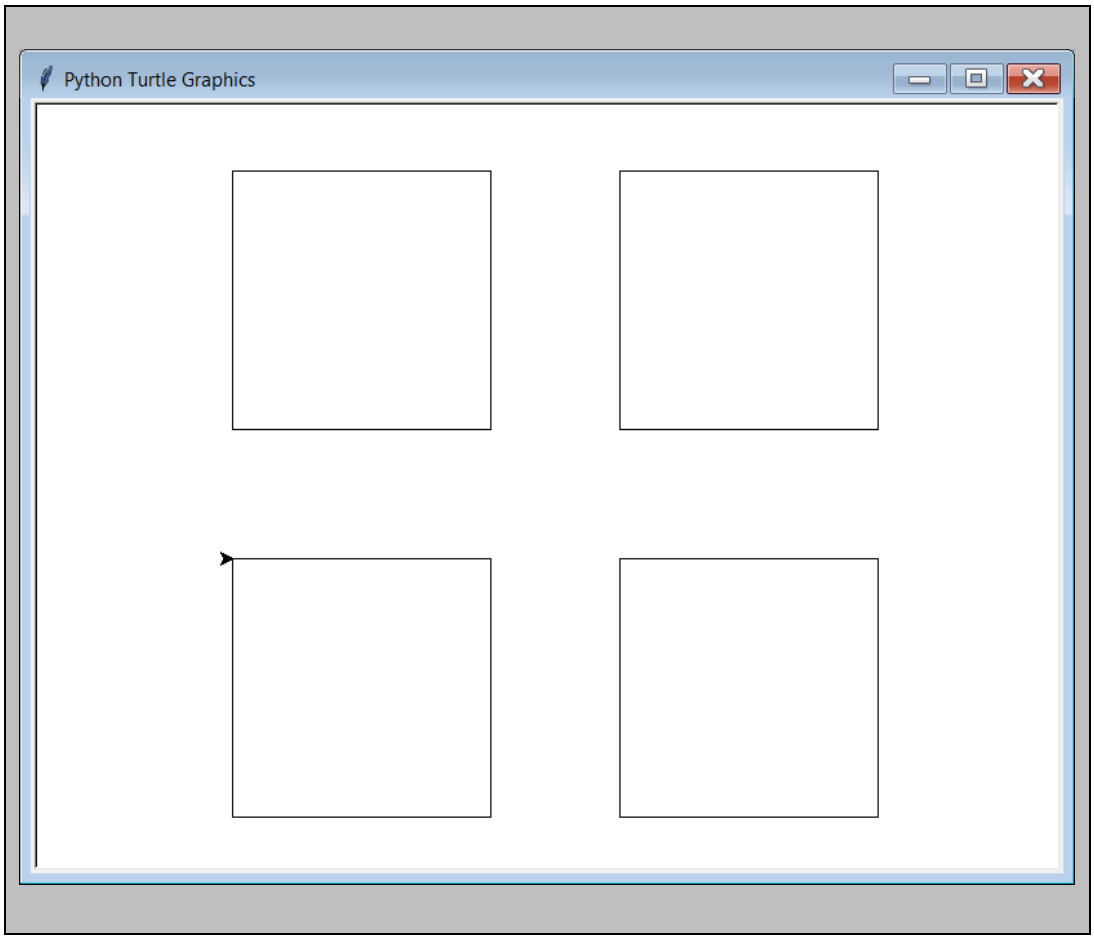
All of the examples that we have seen so far have been *text* programs, but what about *graphics* programs? Program **GraphicsProcedures01.py**, in Figure 9.9, is the first of several graphics program examples. This particular program is actually a copy of **TurtleGraphics03.py** from Chapter V. The program draws 4 squares in 4 different locations. Note how the program is a bit lengthy and

repetitive. This is because the 8 lines of code necessary to display a square are repeated 4 times.

Figure 9.9

```
# GraphicsProcedures01.py
# This is a copy of TurtleGraphics03.py from Chapter 5.
# Note the repetitive code needed to draw the 4 squares.


from turtle import *

setup(800,600)

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

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

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

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

```
35 forward(200)
36 right(90)
37 forward(200)
38 right(90)
39
40 penup()
41 right(90)     # face South
42 forward(300)  # Move to bottom-right corner
43 left(90)      # face East again
44 pendown()
45
46 # draw square
47 forward(200)
48 right(90)
49 forward(200)
50 right(90)
51 forward(200)
52 right(90)
53 forward(200)
54 right(90)
55
56 penup()
57 backward(300) # Move to bottom-left corner
58 pendown()
59
60 # draw square
61 forward(200)
62 right(90)
63 forward(200)
64 right(90)
65 forward(200)
66 right(90)
67 forward(200)
68 right(90)
69
70 update()
71 done()
```



Program **GraphicsProcedures02.py**, in Figure 9.10, has the exact same output as the previous program, but is more efficient because a special **drawSquare** procedure has been defined. Now, instead of repeating the 8 lines of code to draw the square 4 times, it is only written once. The procedure is just called 4 times.

Figure 9.10

```
1 # GraphicsProcedures02.py
2 # This program has the same output as the previous
3 # program but is more efficient because a special
4 # <drawSquare> procedure has been created which
5 # eliminates the repetitive code.
6
7
8 from turtle import *
9
```

```
10
11 def drawSquare():
12     pendown()
13     forward(200)
14     right(90)
15     forward(200)
16     right(90)
17     forward(200)
18     right(90)
19     forward(200)
20     right(90)
21     penup()
22
23
24
25 ##########
26 #  MAIN  #
27 ##########
28
29 setup(800,600)
30
31 penup()
32 left(135)     # face North-West
33 forward(350)  # Move to top-left corner
34 right(135)    # face East again
35
36 drawSquare()
37
38 forward(300)  # Move to top-right corner
39
40 drawSquare()
41
42 right(90)     # face South
43 forward(300)  # Move to bottom-right corner
44 left(90)      # face East again
45
46 drawSquare()
47
```

```
48 backward(300) # Move to bottom-left corner
49
50 drawSquare()
51
52 update()
53 done()
54
```

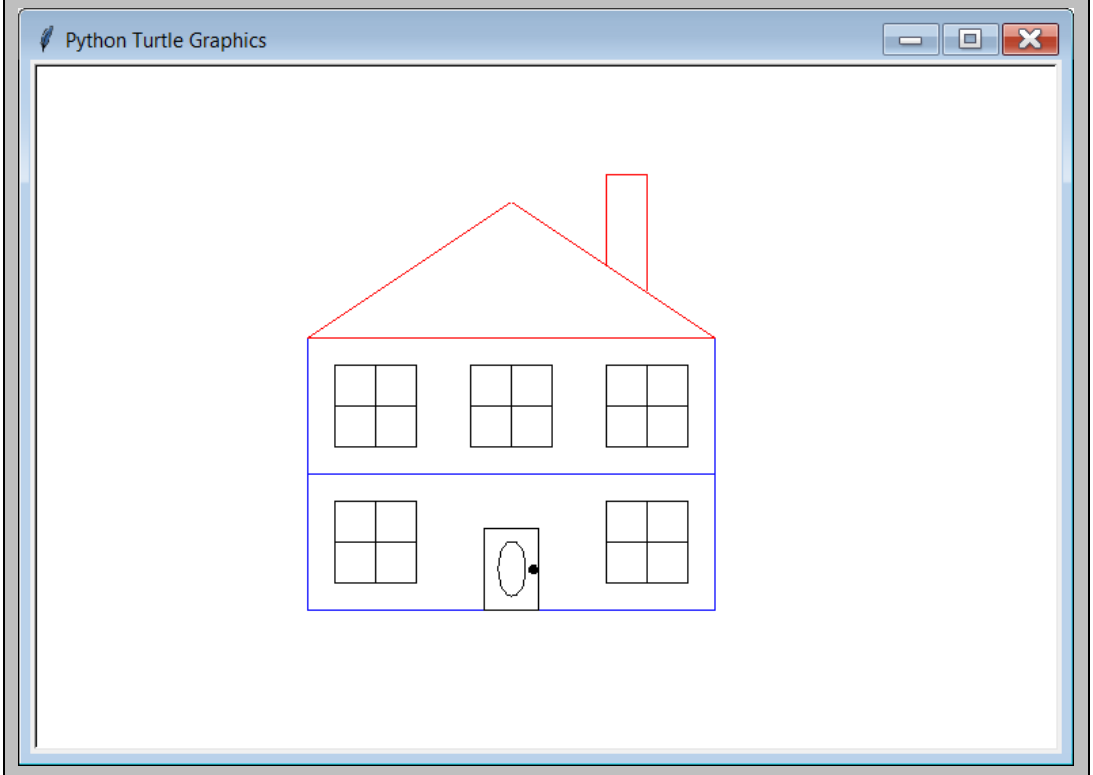
Traditional Graphics Examples

We are now going to switch from *Turtle Graphics* to *Traditional Graphics*. Program **GraphicsProcedures03.py**, in Figure 9.11, starts things off by drawing a house using now procedures at all.

Figure 9.11

```
 1 # GraphicsProcedures03.py
 2 # This program demonstrates drawing a house.
 3 # Imagine that you want to change the appearance
 4 # of the chimney.  What code would you change?
 5
 6
 7 from Graphics import *
 8
 9
10 beginGrfx(750,500)
11
12 setColor("blue")
13 drawRectangle(200,200,500,300)
14 drawRectangle(200,300,500,400)
15 setColor("red")
16 drawLine(200,200,350,100)
17 drawLine(500,200,350,100)
18 drawLine(200,200,500,200)
19 drawLine(420,146,420,80)
20 drawLine(420,80,450,80)
21 drawLine(450,80,450,166)
22 setColor("black")
23 drawRectangle(330,340,370,400)
```

```
24 drawOval(350,370,10,20)
25 fillCircle(366,370,3)
26 drawRectangle(220,220,280,280)
27 drawLine(220,250,280,250)
28 drawLine(250,220,250,280)
29 drawRectangle(420,220,480,280)
30 drawLine(420,250,480,250)
31 drawLine(450,220,450,280)
32 drawRectangle(320,220,380,280)
33 drawLine(320,250,380,250)
34 drawLine(350,220,350,280)
35 drawRectangle(220,320,280,380)
36 drawLine(220,350,280,350)
37 drawLine(250,320,250,380)
38 drawRectangle(420,320,480,380)
39 drawLine(420,350,480,350)
40 drawLine(450,320,450,380)
41
42 endGrfx()
43
```



You may still not appreciate what the point is here. Imagine that you do not like the appearance of the chimney. Where in this program do you go to make the changes? It is not really that easy to see.

I hope you understand that the house displayed in Figure 9.11 can easily be made far more complex with hundreds of program statements. Just imagine if the house includes actual bricks, bushes, plants in the windows, a number on the doors, Christmas lights, and kids playing in the yard. In no time a project as the one described requires more than 1000 program statements. Placing all those statements in the "**MAIN**" section of your program without organizing them into separate procedures is very poor program design. Making any changes becomes very tedious. Even with this relatively small program could you tell just by the program statements that they will draw a house?

Sometimes when students work on a program that contains too many lines in one module, they argue that they know the purpose of each program statement. This is quite true when the program writing is fresh in your mind. It is a different story when time goes by. We wrote an Academic Decathlon program of about 12,000 lines about twenty years ago. We were in a hurry to meet a deadline. The program lacked proper modular program design and comments were pretty much non-existent. In the heat of writing the program day after day, we stayed on top of everything. Then we received a wake-up call one year later. It was a program for Academic Decathlon data processing and there were major changes. Altering the poorly-designed program a year later was a nightmare. Accept it. The rules of "Proper Programming Design" are important.

Imagine you need to write a major research paper for English class. Do you begin by typing the final copy? Hopefully not. Most English teachers will require you to first turn in an *outline*. It is similar with a program. If I want to draw a house, I would start by asking myself "What is needed to draw a house?" I would then answer myself, "A house has a door, roof, chimney, windows and (since this will be a 2-story house) a couple floors.

Now look at **GraphicsProcedures04.py**, in Figure 9.12. You will see that the house program is now divided up into 5 different procedures: **drawFloors**, **drawRoof**, **drawDoor**, **drawWindows** and **drawChimney**. Look closely at the "**MAIN**" section of the program. Doesn't it look like an outline? This is what the "**MAIN**" section is supposed to look like in a well-designed program.

Figure 9.12

```
1 # GraphicsProcedures04.py
2 # This program divides the house up into several
3 # different procedures.  Imagine again that you
4 # want to change the appearance of the chimney.
```

```
 5 # Finding the appropriate code is much easier now.
 6 # NOTE: The "MAIN" section of the program now looks
 7 #       like an outline.
 8
 9
10 from Graphics import *
11
12
13 def drawFloors():
14     setColor("blue")
15     drawRectangle(200,200,500,300)
16     drawRectangle(200,300,500,400)
17
18
19 def drawRoof():
20     setColor("red")
21     drawLine(200,200,350,100)
22     drawLine(500,200,350,100)
23     drawLine(200,200,500,200)
24
25
26 def drawChimney():
27     drawLine(420,146,420,80)
28     drawLine(420,80,450,80)
29     drawLine(450,80,450,166)
30
31
32 def drawDoor():
33     setColor("black")
34     drawRectangle(330,340,370,400)
35     drawOval(350,370,10,20)
36     fillCircle(366,370,3)
37
38
39 def drawWindows():
40     drawRectangle(220,220,280,280)
41     drawLine(220,250,280,250)
42     drawLine(250,220,250,280)
43     drawRectangle(420,220,480,280)
44     drawLine(420,250,480,250)
45     drawLine(450,220,450,280)
46     drawRectangle(320,220,380,280)
47     drawLine(320,250,380,250)
48     drawLine(350,220,350,280)
49     drawRectangle(220,320,280,380)
50     drawLine(220,350,280,350)
```

```
51      drawLine(250,320,250,380)
52      drawRectangle(420,320,480,380)
53      drawLine(420,350,480,350)
54      drawLine(450,320,450,380)
55
56
57
58  ##########
59  #  MAIN  #
60  ##########
61
62  beginGrfx(750,500)
63
64  drawFloors()
65  drawRoof()
66  drawChimney()
67  drawDoor()
68  drawWindows()
69
70  endGrfx()
71
```

Consider the chimney question that was asked with the previous program. If this program's chimney had issues you would go to the **drawChiney** procedure to fix it. If you did not like the windows, you would go to **drawWindows**. Having procedures with *self-commenting* names makes debugging and enhancing your programs much simpler.

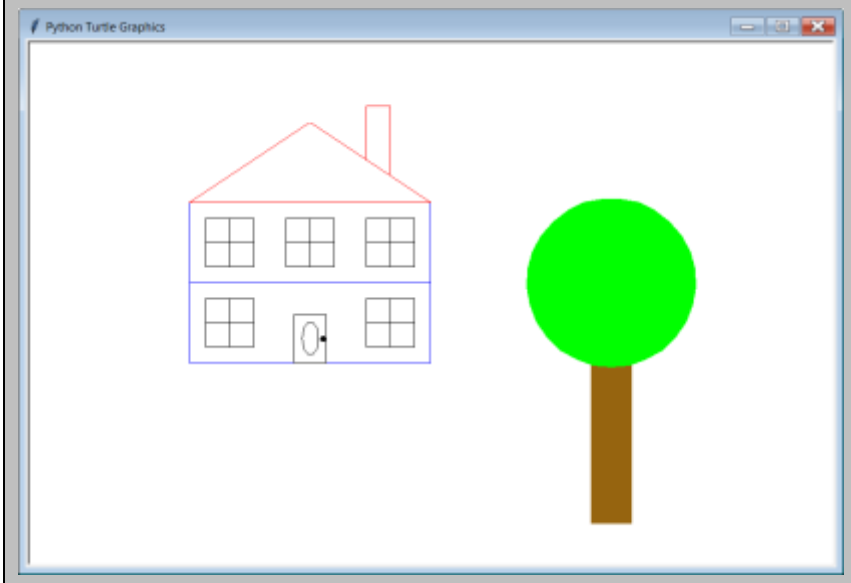
Now, what happens if you are drawing more than a house? What if the house a *tree* in the front yard? Program **GraphicsProcedures05.py**, in Figure 9.13, adds a couple procedures called **drawTrunk** and **drawLeaves**. Note that both procedures are called from the "**MAIN**" section of the program.

Figure 9.13

```
1  # GraphicsProcedures05.py
2  # This program adds 2 more procedures for drawing a tree.
3
4
5  from Graphics import *
6
7
```

```
 8 # House procedures
 9
10 def drawFloors():
11     setColor("blue")
12     drawRectangle(200,200,500,300)
13     drawRectangle(200,300,500,400)
14
15
16 def drawRoof():
17     setColor("red")
18     drawLine(200,200,350,100)
19     drawLine(500,200,350,100)
20     drawLine(200,200,500,200)
21
22
23 def drawChimney():
24     drawLine(420,146,420,80)
25     drawLine(420,80,450,80)
26     drawLine(450,80,450,166)
27
28
29 def drawDoor():
30     setColor("black")
31     drawRectangle(330,340,370,400)
32     drawOval(350,370,10,20)
33     fillCircle(366,370,3)
34
35
36 def drawWindows():
37     drawRectangle(220,220,280,280)
38     drawLine(220,250,280,250)
39     drawLine(250,220,250,280)
40     drawRectangle(420,220,480,280)
41     drawLine(420,250,480,250)
42     drawLine(450,220,450,280)
43     drawRectangle(320,220,380,280)
44     drawLine(320,250,380,250)
45     drawLine(350,220,350,280)
46     drawRectangle(220,320,280,380)
47     drawLine(220,350,280,350)
48     drawLine(250,320,250,380)
49     drawRectangle(420,320,480,380)
50     drawLine(420,350,480,350)
51     drawLine(450,320,450,380)
52
53
```

```
54 # Tree procedures
55
56 def drawTrunk():
57     setColor("brown")
58     fillRectangle(700,400,750,600)
59
60
61 def drawLeaves():
62     setColor("green")
63     fillCircle(725,300,105)
64
65
66
67 ##########
68 #  MAIN  #
69 ##########
70
71 beginGrfx(1000,650)
72
73 drawFloors()
74 drawRoof()
75 drawChimney()
76 drawDoor()
77 drawWindows()
78
79 drawTrunk()
80 drawLeaves()
81
82 endGrfx()
```



Now suppose we want to add a simple background to this program. Program **GraphicsProcedures06.py**, in Figure 9.14, adds procedures **drawSky** and **drawGrass**. When you run the program, you will no longer see the house and tree. The *house* and *tree* procedures are still in the program and technically they were drawn. You do not see them because the background was drawn *on top of* them. This happened because the background was drawn *after* the house and the tree. This is a common mistake made by students who sometimes argue that their program *should* work because the *background* procedures are before the *house* and *tree* procedures in the program. What must be understood is that the order of the procedure definitions is completely irrelevant. The only thing that matters is the order of the procedure calls in the "**MAIN**" section of the program. If you at this program's "**MAIN**" section, you will see that **drawSky** (which draws a big dark blue rectangle covering the top half of the screen) and **drawGrass** (which draws a big dark green rectangle covering the lower half of the screen) are called last.

Figure 9.14

```
 1 # GraphicsProcedures06.py
 2 # This program adds 2 more procedures for drawing a
 3 # background.  When you run the program, all you see is
 4 # the background.  This is because the background was
 5 # drawn after, and therefore on top of, everything else.
 6
 7
 8 from Graphics import *
 9
10
11 # Background procedures
12
13 def drawSky():
14     setColor("sky blue")
15     fillRectangle(0,0,1000,325)
16
17
18 def drawGrass():
19     setColor("dark green")
20     fillRectangle(0,325,1000,650)
21
22
23 # House procedures
24
```

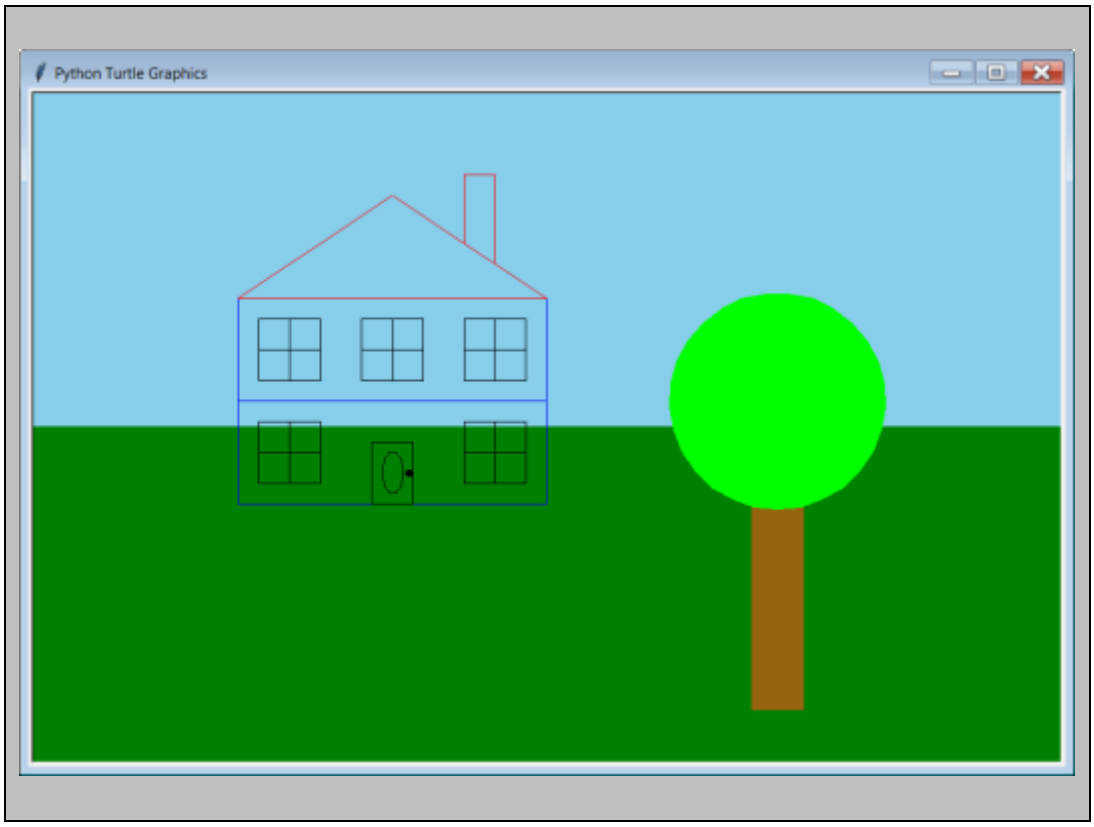
```
25 def drawFloors():
26     setColor("blue")
27     drawRectangle(200,200,500,300)
28     drawRectangle(200,300,500,400)
29
30
31 def drawRoof():
32     setColor("red")
33     drawLine(200,200,350,100)
34     drawLine(500,200,350,100)
35     drawLine(200,200,500,200)
36
37
38 def drawChimney():
39     drawLine(420,146,420,80)
40     drawLine(420,80,450,80)
41     drawLine(450,80,450,166)
42
43
44 def drawDoor():
45     setColor("black")
46     drawRectangle(330,340,370,400)
47     drawOval(350,370,10,20)
48     fillCircle(366,370,3)
49
50
51 def drawWindows():
52     drawRectangle(220,220,280,280)
53     drawLine(220,250,280,250)
54     drawLine(250,220,250,280)
55     drawRectangle(420,220,480,280)
56     drawLine(420,250,480,250)
57     drawLine(450,220,450,280)
58     drawRectangle(320,220,380,280)
59     drawLine(320,250,380,250)
60     drawLine(350,220,350,280)
61     drawRectangle(220,320,280,380)
62     drawLine(220,350,280,350)
63     drawLine(250,320,250,380)
64     drawRectangle(420,320,480,380)
65     drawLine(420,350,480,350)
66     drawLine(450,320,450,380)
67
68
69 # Tree procedures
70
71 def drawTrunk():
72     setColor("brown")
73     fillRectangle(700,400,750,600)
74
75
```

```
76 def drawLeaves():
77     setColor("green")
78     fillCircle(725,300,105)
79
80
81
82 ##########
83 #  MAIN  #
84 ##########
85
86 beginGrfx(1000,650)
87
88 drawFloors()
89 drawRoof()
90 drawChimney()
91 drawDoor()
92 drawWindows()
93 drawTrunk()
94 drawLeaves()
95
96 drawSky()
97 drawGrass()
98
99 endGrfx()
```

Program **GraphicsProcedures07.py**, in Figure 9.15, fixes the problem of the previous program by calling the 2 *background* procedures first before anything else is drawn. The house and the tree are drawn after, and therefore on top of, the background. While the tree looks fine, there is 1 remaining problem with the house. It is transparent. You can see both the sky and the grass through the house. This is because unlike the tree, the house is not *solid*.

Figure 9.15

```
  1 # GraphicsProcedures07.py
  2 # This program fixes the issue of the previous
  3 # program by drawing the background first.
  4 # The house does not look right because we
  5 # can see the background through it.
  6 # Unlike the tree, the house is not solid.
  7

    :    :    :    :    :    :    :    :

 82
 83 ###########
 84 #  MAIN   #
 85 ###########
 86
 87 beginGrfx(1000,650)
 88
 89 drawSky()
 90 drawGrass()
 91
 92 drawFloors()
 93 drawRoof()
 94 drawChimney()
 95 drawDoor()
 96 drawWindows()
 97 drawTrunk()
 98 drawLeaves()
 99
100 endGrfx()
101
```

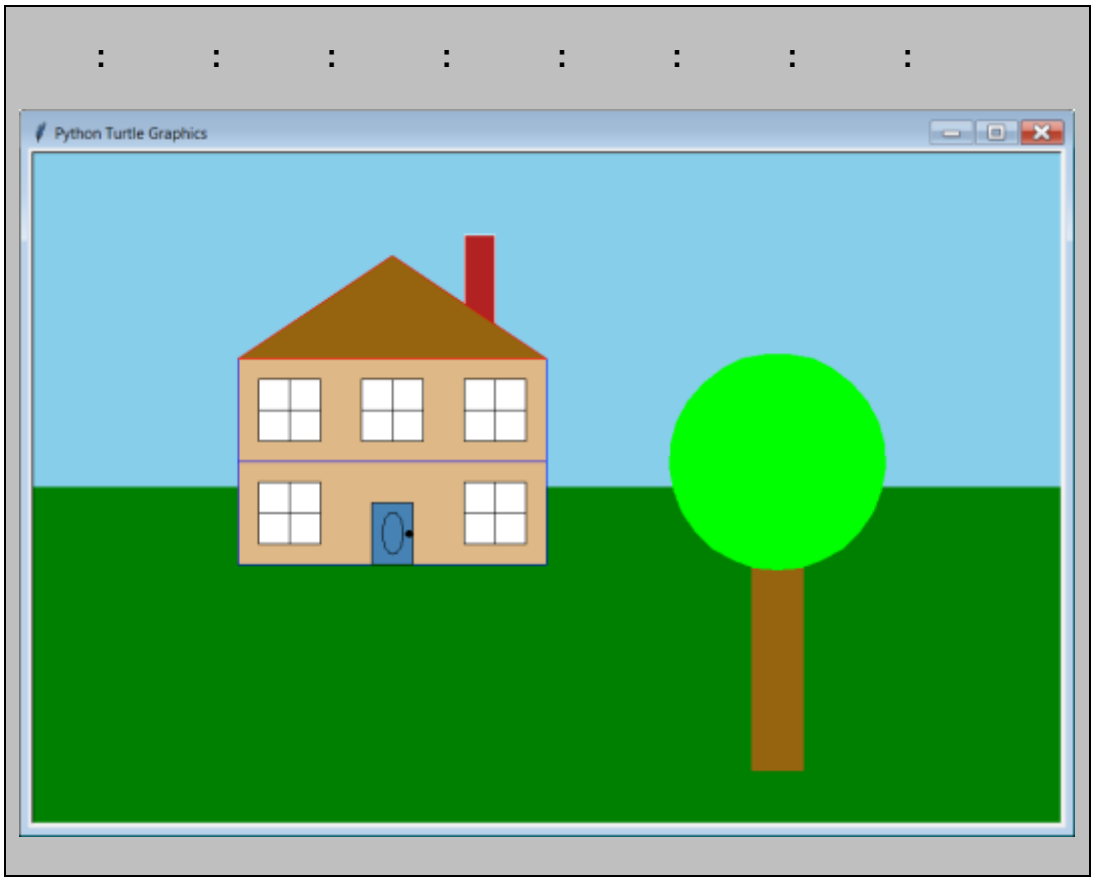


Program **GraphicsProcedures08.py**, in Figure 9.16, changes the *house* procedures so they draw *solid* shapes. Now the output appears as it was intended.

Figure 9.16

```
 1 # GraphicsProcedures08.py
 2 # This program makes the house solid by replacing
 3 # several "draw" commands with "fill" commands.
 4

   :     :     :     :     :     :     :     :

30
31 # House procedures
32
33 def drawFloors():
34     setColor("burlywood")
35     fillRectangle(200,200,500,400)
36     setColor("blue")
37     drawRectangle(200,200,500,300)
38     drawRectangle(200,300,500,400)
39
```

```
def drawRoof():
    setColor("brown")
    fillPolygon([200,200,350,100,500,200])
    setColor("red")
    drawLine(200,200,350,100)
    drawLine(500,200,350,100)
    drawLine(200,200,500,200)

def drawChimney():
    setColor("firebrick")
    fillPolygon([420,146,420,80,450,80,450,166])
    setColor("white")
    drawLine(420,146,420,80)
    drawLine(420,80,450,80)
    drawLine(450,80,450,166)

def drawDoor():
    setColor("steel blue")
    fillRectangle(330,340,370,400)
    setColor("black")
    drawRectangle(330,340,370,400)
    drawOval(350,370,10,20)
    fillCircle(366,370,3)

def drawWindows():
    setColor("white")
    fillRectangle(220,220,280,280)
    fillRectangle(420,220,480,280)
    fillRectangle(320,220,380,280)
    fillRectangle(220,320,280,380)
    fillRectangle(420,320,480,380)
    setColor("black")
    drawRectangle(220,220,280,280)
    drawLine(220,250,280,250)
    drawLine(250,220,250,280)
    drawRectangle(420,220,480,280)
    drawLine(420,250,480,250)
    drawLine(450,220,450,280)
    drawRectangle(320,220,380,280)
    drawLine(320,250,380,250)
    drawLine(350,220,350,280)
    drawRectangle(220,320,280,380)
    drawLine(220,350,280,350)
    drawLine(250,320,250,380)
    drawRectangle(420,320,480,380)
    drawLine(420,350,480,350)
    drawLine(450,320,450,380)
```

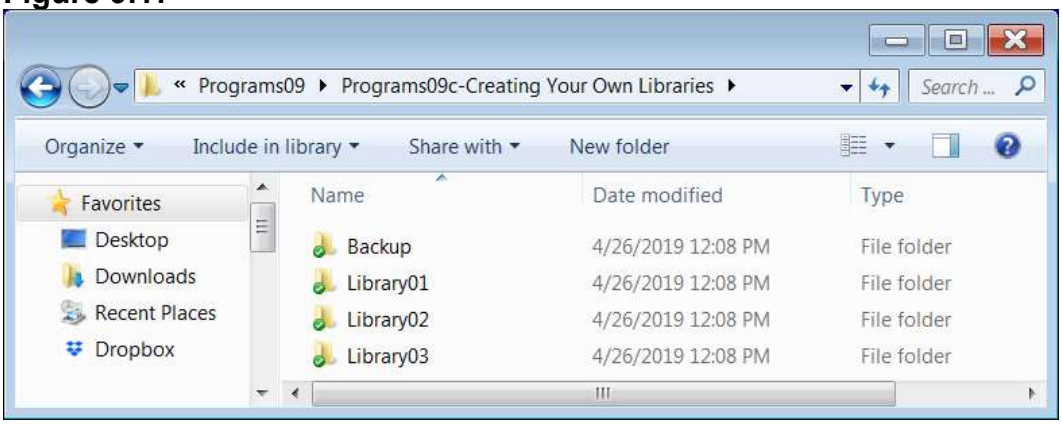


9.5 Creating Your Own Libraries

So far, in this class, we have discussed a number of libraries like **turtle**, **math**, **time**, **random** and **Graphics**. Keep in mind that you are not limited to these libraries. Aside from the many other libraries that are built into Python, you have the ability to create your own. If that were not possible, I would never have been able to create the **Graphics** library in the first place.

When you open the **Programs09c-Creating Your Own Libraries** folder, as shown in Figure 9.17, you will see that it is different from the other subfolders in **Programs09**. Besides the familiar **Backup** folder, this subfolder simply contains 3 other subfolders.

Figure 9.17



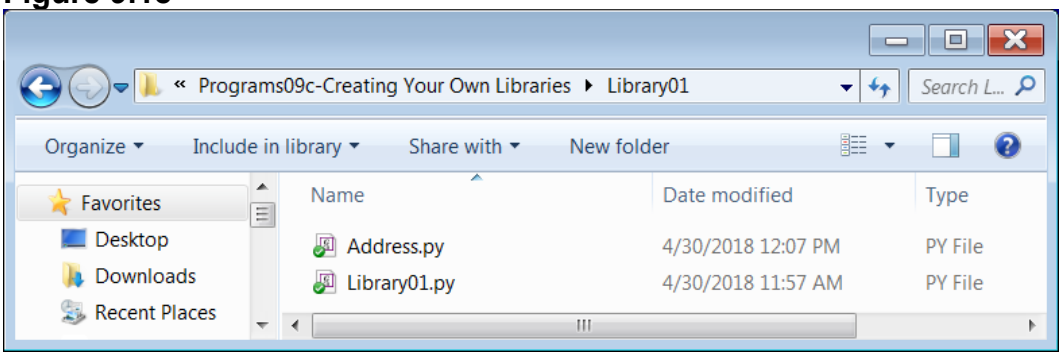
Since this section is on creating your own libraries, it means that each program example will actually consist of multiple files.

NOTE: Whenever a program consists of more than one file, all of the necessary files for that program should be placed into a single folder.

ALSO: Whenever a program consists of more than one file, the file that contains the "**MAIN**" section of the program is called the "Driving File". It is this file that needs to be executed.

Figure 9.18 shows the contents of subfolder **Library01**. There are 2 files: **Library01.py** and **Address.py**. The *driving file* is **Library01.py** so we will load that file first.

Figure 9.18



When you look at program **Library01.py**, shown in Figure 9.19, you see that it is very similar to program **SimpleProcedures02.py**. The difference is while there are procedure calls, there are no procedure definitions. Somehow, this program still manages to execute.

Figure 9.19

```
1 # Library01.py
2 # This program has the same output as SimpleProcedures02.py
3 # The difference is all of the procedures are now in a
4 # user-created library called <Address> which is located
5 # in the file Address.py
6
7
8 from Address import *
9
10
11 displayName()
12 displayStreetAddress()
13 displayCityStateZip()
14
```

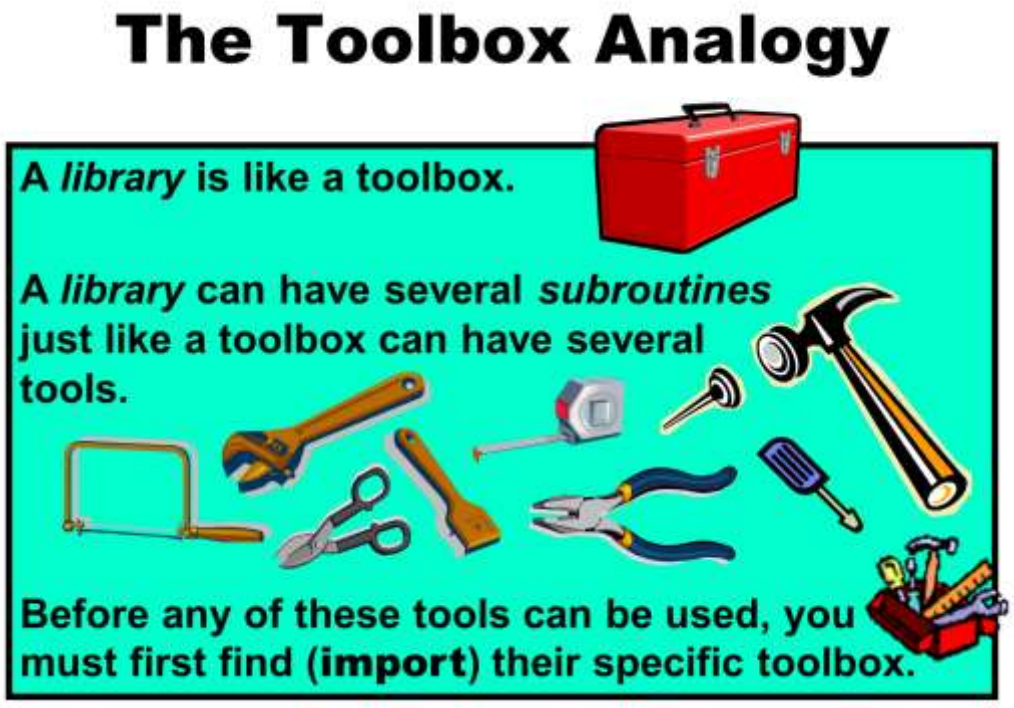
```
----jGRASP exec: python Library01.py

Kathy Smith
7003 Orleans Court
Kensington, MD 20795

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

So how is this possible? Look at line 8. Notice the command: **from Address import ***. This is how we have been importing libraries like **turtle**, **math**, **time**, **random** and **Graphics**. When you create the library yourself, it is imported in the exact same manner. For this to work, it does assume the **Address** library, which is the file **Address.py**, is located in the same folder. This file is shown in Figure 9.20. If you wish, you can click the **Run** icon in this program, but nothing will execute. The only purpose in doing this is that it would let you know if you had any syntax errors. Then again, it would do the same thing if you just ran the driving file like you are supposed to.

Figure 9.20

```
 1 # Address.py
 2 # This is the <Address> library used by Library01.py
 3 # NOTE: When creating a library, you can click "Run"
 4 # to see if there are any syntax errors.  Keep in mind,
 5 # the program will not actually execute because it has
 6 # no "MAIN" section and is not the "Driving File".
 7
 8
 9 def displayName():
10     print()
11     print("Kathy Smith")
12
13
14 def displayStreetAddress():
15     print("7003 Orleans Court")
16
17
18 def displayCityStateZip():
19     print("Kensington, MD 20795")
20
```

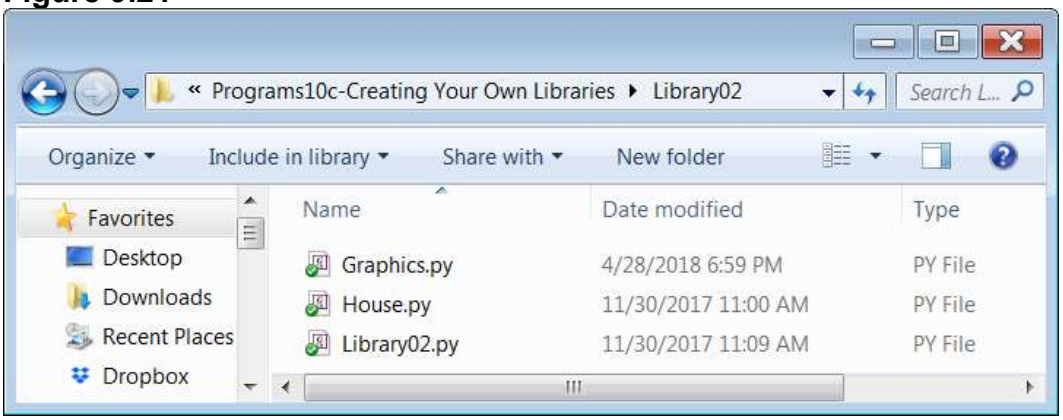
Remember…



There are two benefits to putting your subroutines inside libraries in this manner. First, the driving file is less cluttered. Second, you can import these subroutines from other programs as well. Think of the trusty toolbox analogy. If we have created a useful tool, we do not just throw it away when the job is done. We put it in a toolbox so is can be used on a future project. Come to think of it, isn't this exactly what we have been doing with the **Graphics** library?

Figure 9.21 shows the contents of the next subfolder, **Library02**. This example has 3 files: There is the driving file, **Library02.py**; there is the file for the **House** library, **House.**py; and since this is a graphics program, we also need the file for the **Graphics** library, **Graphics.py**.

Figure 9.21

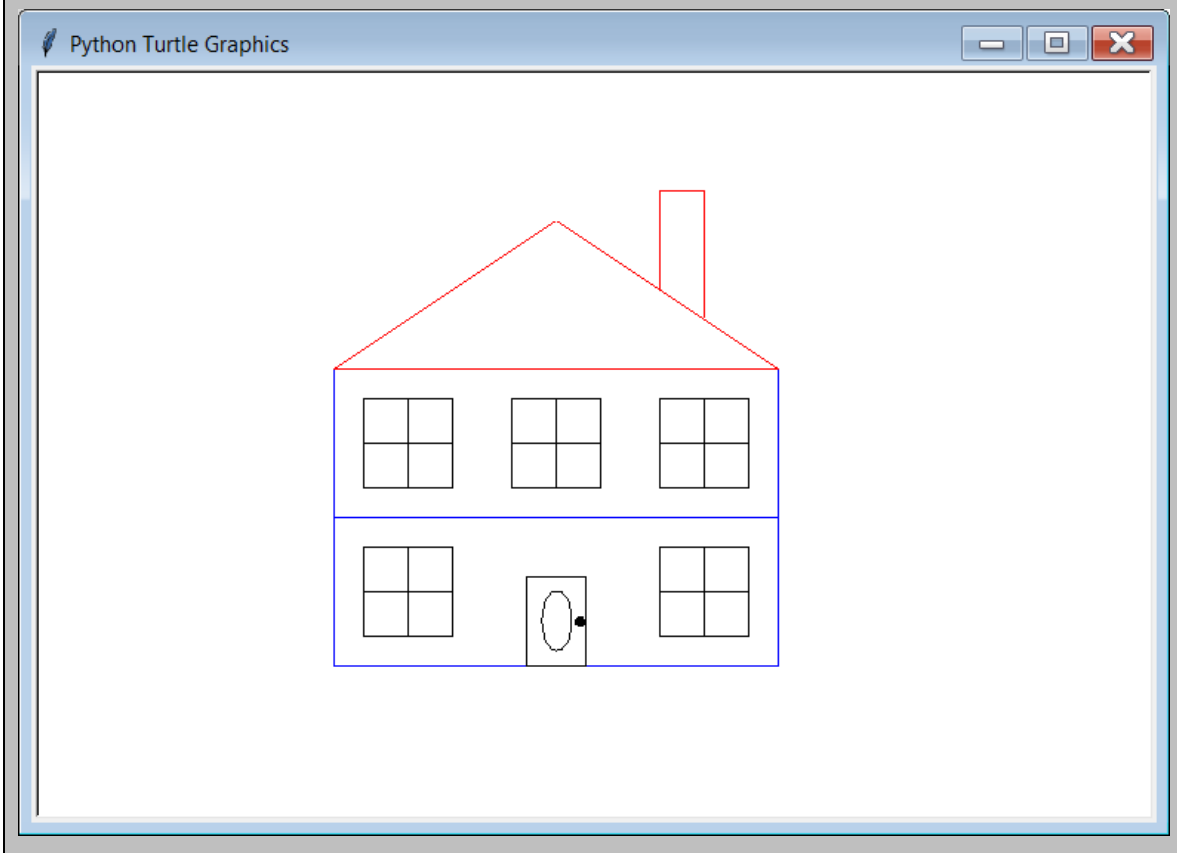


When you look at program **Library02.py**, shown in Figure 9.22, you see that it is very similar to program **GraphicsProcedures04.py**, which drew a simple house. As with the last example, we see again that while there are procedure calls, there are no procedure definitions. These are imported from the **House** library shown in Figure 9.23. Note that one program can import multiple libraries.

Figure 9.22

```
1 # Library02.py
2 # This program has the same output as GraphicsProcedures04.py
3 # The difference is all of the procedures are now in a
4 # user-created library called <House> which is located
5 # in the file House.py
6
7 # NOTE: Technically, we do not need to import <Graphics>
8 # because it is already imported by the <House> library.
9
```

```
10
11 from Graphics import *
12 from House import *
13
14
15 beginGrfx(750,500)
16
17 drawFloors()
18 drawRoof()
19 drawChimney()
20 drawDoor()
21 drawWindows()
22
23 endGrfx()
```



Notice also that one library can import another. In this case, the **House** library imports the **Graphics** library. Truth be told, the **Graphics** library imports the **turtle** library and number of other libraries as well.

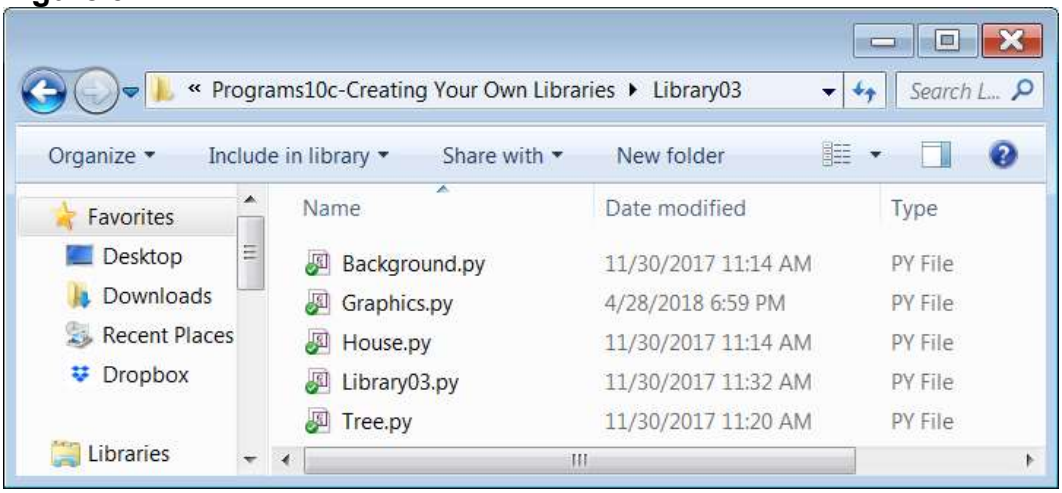
Figure 9.23

```
 1 # House.py
 2 # This is the <House> library used by Library02.py
 3
 4 from Graphics import *
 5
 6 def drawFloors():
 7    setColor("blue")
 8    drawRectangle(200,200,500,300)
 9    drawRectangle(200,300,500,400)
10
11 def drawRoof():
12    setColor("red")
13    drawLine(200,200,350,100)
14    drawLine(500,200,350,100)
15    drawLine(200,200,500,200)
16
17 def drawChimney():
18    drawLine(420,146,420,80)
19    drawLine(420,80,450,80)
20    drawLine(450,80,450,166)
21
22 def drawDoor():
23    setColor("black")
24    drawRectangle(330,340,370,400)
25    drawOval(350,370,10,20)
26    fillCircle(366,370,3)
27
28 def drawWindows():
29    drawRectangle(220,220,280,280)
30    drawLine(220,250,280,250)
31    drawLine(250,220,250,280)
32    drawRectangle(420,220,480,280)
33    drawLine(420,250,480,250)
34    drawLine(450,220,450,280)
35    drawRectangle(320,220,380,280)
36    drawLine(320,250,380,250)
37    drawLine(350,220,350,280)
```

```
38      drawRectangle(220,320,280,380)
39      drawLine(220,350,280,350)
40      drawLine(250,320,250,380)
41      drawRectangle(420,320,480,380)
42      drawLine(420,350,480,350)
43      drawLine(450,320,450,380)
```

Figure 9.24 shows the contents of the third and final subfolder, **Library03**. This example has 5 separate files: There is the driving file, **Library03.py**; the **House** library, **House.**py; and the **Graphics** library, **Graphics.py**. This time, there are 2 additional library files, **Background.py** and **Tree.py**.

Figure 9.24



When you look at program **Library02.py**, shown in Figure 9.25, you see that it is very similar to program **GraphicsProcedures08.py**, which was the last of the "house drawing" examples. In addition to the house (that is not transparent), there is a tree and a background. As with the previous 2 examples, we see yet again that while there are procedure calls, there are no procedure definitions. These are imported from the **House**, **Background**, and **Tree** libraries shown in Figure 9.26, 9.27 and 9.28 respectively.

Figure 9.25

```
1 # Library03.py
2 # This program has the same output as GraphicsProcedures08.py
3 # It demonstrates that multiple user-created libraries can
4 # be imported from the same file.  Breaking up a program in
5 # this manner makes things more organized and less cluttered.
6
```

```
 7
 8 from Graphics import *
 9 from House import *
10 from Tree import *
11 from Background import *
12
13
14 beginGrfx(1000,650)
15
16 drawSky()
17 drawGrass()
18 drawFloors()
19 drawRoof()
20 drawChimney()
21 drawDoor()
22 drawWindows()
23 drawTrunk()
24 drawLeaves()
25
26 endGrfx()
```

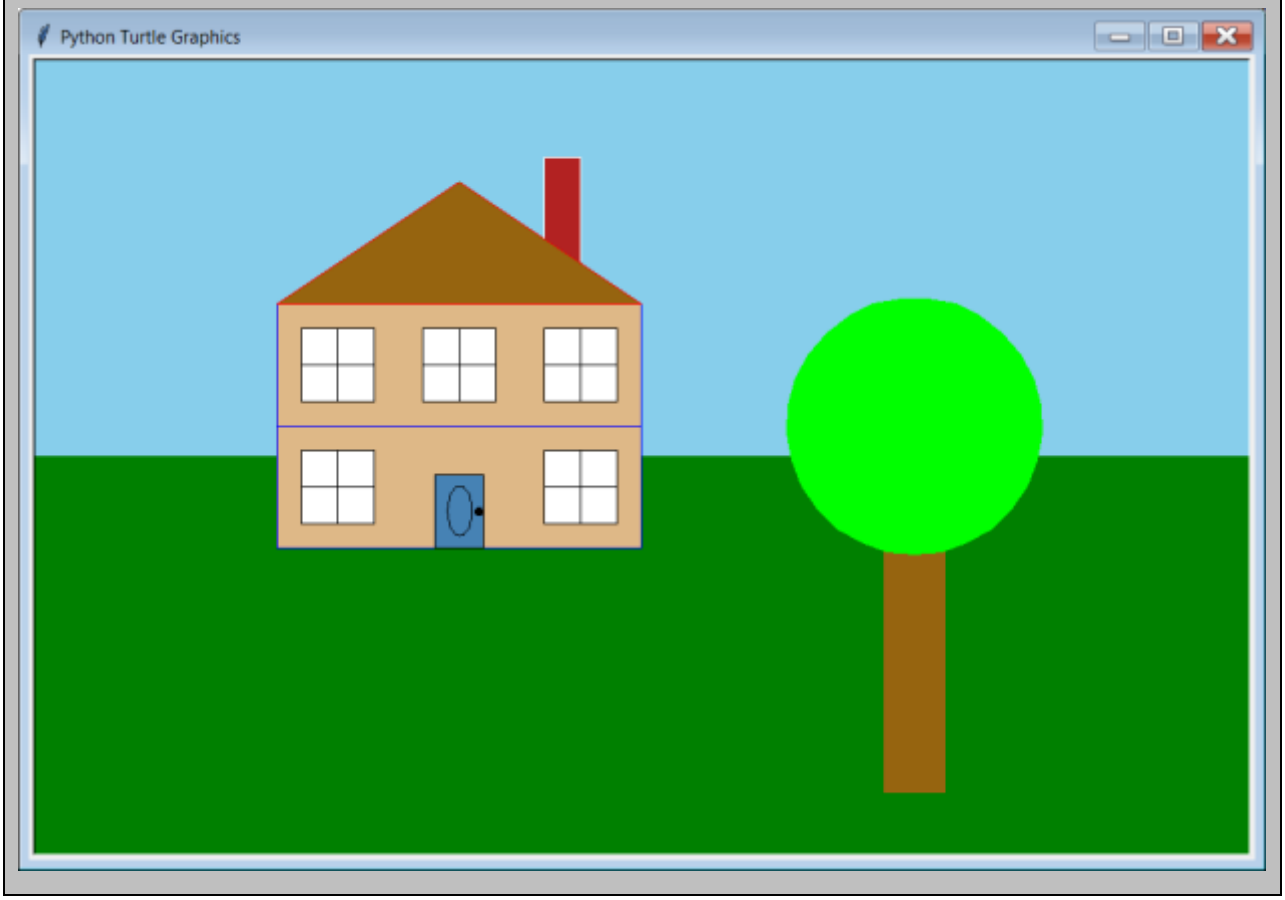


Figure 9.26

```
# House.py
# This is the <House> library used by Library03.py


from Graphics import *


def drawFloors():
    setColor("burlywood")
    fillRectangle(200,200,500,400)
    setColor("blue")
    drawRectangle(200,200,500,300)
    drawRectangle(200,300,500,400)


def drawRoof():
    setColor("brown")
    fillPolygon([200,200,350,100,500,200])
    setColor("red")
    drawLine(200,200,350,100)
    drawLine(500,200,350,100)
    drawLine(200,200,500,200)


def drawChimney():
    setColor("firebrick")
    fillPolygon([420,146,420,80,450,80,450,166])
    setColor("white")
    drawLine(420,146,420,80)
    drawLine(420,80,450,80)
    drawLine(450,80,450,166)


def drawDoor():
    setColor("steel blue")
    fillRectangle(330,340,370,400)
    setColor("black")
    drawRectangle(330,340,370,400)
    drawOval(350,370,10,20)
    fillCircle(366,370,3)


def drawWindows():
```

```
44     setColor("white")
45     fillRectangle(220,220,280,280)
46     fillRectangle(420,220,480,280)
47     fillRectangle(320,220,380,280)
48     fillRectangle(220,320,280,380)
49     fillRectangle(420,320,480,380)
50     setColor("black")
51     drawRectangle(220,220,280,280)
52     drawLine(220,250,280,250)
53     drawLine(250,220,250,280)
54     drawRectangle(420,220,480,280)
55     drawLine(420,250,480,250)
56     drawLine(450,220,450,280)
57     drawRectangle(320,220,380,280)
58     drawLine(320,250,380,250)
59     drawLine(350,220,350,280)
60     drawRectangle(220,320,280,380)
61     drawLine(220,350,280,350)
62     drawLine(250,320,250,380)
63     drawRectangle(420,320,480,380)
64     drawLine(420,350,480,350)
65     drawLine(450,320,450,380)
66
```

Figure 9.27

```
 1 # Background.py
 2 # This is the <Background> library used by Library03.py
 3
 4
 5 from Graphics import *
 6
 7
 8 def drawSky():
 9     setColor("sky blue")
10     fillRectangle(0,0,1000,325)
11
12
13 def drawGrass():
14     setColor("dark green")
15     fillRectangle(0,325,1000,650)
16
```

Figure 9.28

```
 1 # Tree.py
 2 # This is the <Tree> library used by Library03.py
 3
 4
 5 from Graphics import *
 6
 7
 8 def drawTrunk():
 9    setColor("brown")
10    fillRectangle(700,400,750,600)
11
12
13 def drawLeaves():
14    setColor("green")
15    fillCircle(725,300,105)
16
```

Some Program Design Notes

Programs should not be written by placing all the program statements in the "**MAIN**" section of the program.

Program statements that perform a specific purpose should be placed inside their own module.

These modules are called *procedures* or *functions* in Python.

Good Program Design continues by placing subroutines of a common nature in a separate file, which creates a *library* that can be imported by multiple programs.

9.6 Creating a Big Graphics Program Step-By-Step

You are about to begin **Lab 10A** which is another big graphics project. For this assignment, it will be required that you break the program up into different procedures. There is a step-by-step process which is used to create programs. This will be demonstrated in the next 5 examples. I STRONGLY advise you to use this approach when doing your graphics projects.

Step 1 – Create the "MAIN" section of your program

If you were writing a research paper, your teacher would probably require you to first create an outline. We mentioned earlier that in a well-designed program, the "**MAIN**" section of your program looks like an outline. So this is a good place to start. Program **BigGraphicsStep01.py**, shown in Figure 9.29, starts by creating "**MAIN**" section for a program to draw a house. It should be understood that this first step will not execute. The "**MAIN**" section is trying to call procedures **drawFloors**, **drawRoof**, **drawDoor**, **drawWindows**, and **drawChimney** -- none of which exist.

Figure 9.29

```
 1 # BigGraphicsStep01.py
 2 # Creating a Big Graphics Program
 3 # Step 1 - Create the "MAIN" section of the program.
 4 # This should simply be a list of procedure calls
 5 # and look like an outline.
 6
 7 # NOTE: This first step will not execute because
 8 #       the program is attempting to call procedures
 9 #       which do not exist yet.
10
11
12 from Graphics import *
13
14
15 ###########
16 #  MAIN   #
17 ###########
18
```

```
19 beginGrfx(750,500)
20
21 drawFloors()
22 drawRoof()
23 drawChimney()
24 drawDoor()
25 drawWindows()
26
27 endGrfx()
28
```

```
 ----jGRASP exec: python BigGraphicsStep01.py
Traceback (most recent call last):
  File "BigGraphicsStep01.py", line 21, in <module>
    drawFloors()
NameError: name 'drawFloors' is not defined

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

Step 2 – Create Stubs.

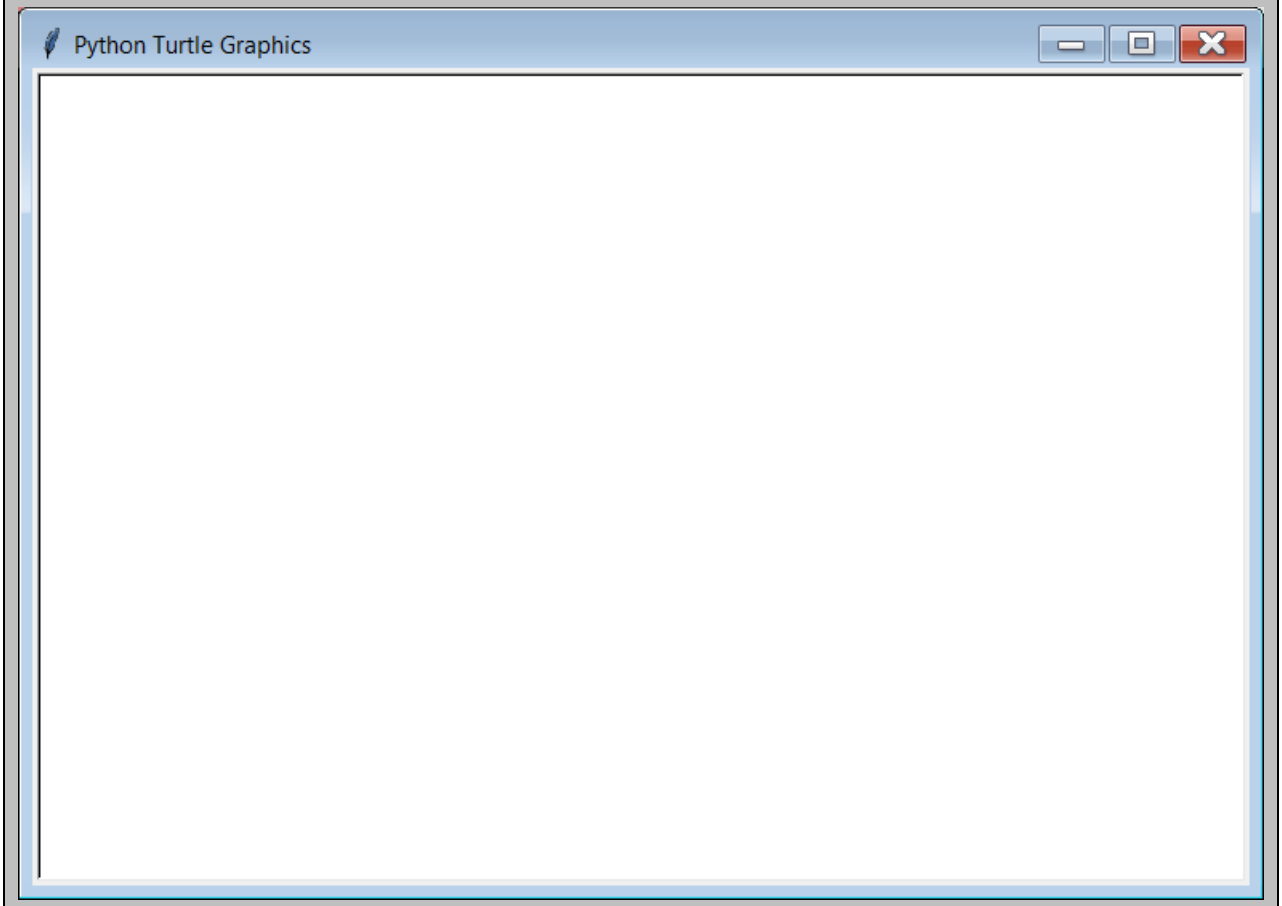
In general terms, a *stub* is a procedure with a proper heading, but no program statements. If we are talking specifically about Python however, this will not work because Python procedures must have at least one indented program statement. The solution is to use the **pass** command. This command actually does nothing. The computer just *passes* over it. Its sole purpose is to take up space temporarily until you are ready to write the actual code of the procedure.

So why are we messing with "*stubs*?" Stubs allow a program to execute without actually writing out all of the procedures. Program **BigGraphicsStep02.py**, shown in Figure 9.30 demonstrates that with stubs in place the program can execute, but it has no output. This is because all of the procedures are essentially empty.

Figure 9.30

```
 1 # BigGraphicsStep02.py
 2 # Creating a Big Graphics Program
 3 # Step 2 - Create Stubs.
 4 # In computer programming, a "stub" is an empty procedure.
 5 # With the stubs in place, the program can now execute.
 6 # Keep in mind that since the program contains nothing but
 7 # stubs, the output will just be a blank white window.
 8
 9 # NOTE: Since Python requires at least one program command
10 #       in each procedure, we will use <pass>.  This is a
11 #       "null program statement" which simply does nothing.
12
13
14 from Graphics import *
15
16
17 def drawFloors():
18     pass
19
20
21 def drawRoof():
22     pass
23
24
25 def drawChimney():
26     pass
27
28
29 def drawDoor():
30     pass
31
32
33 def drawWindows():
34     pass
35
36
```

```
37
38 ##########
39 #  MAIN  #
40 ##########
41
42 beginGrfx(750,500)
43
44 drawFloors()
45 drawRoof()
46 drawChimney()
47 drawDoor()
48 drawWindows()
49
50 endGrfx()
51
```



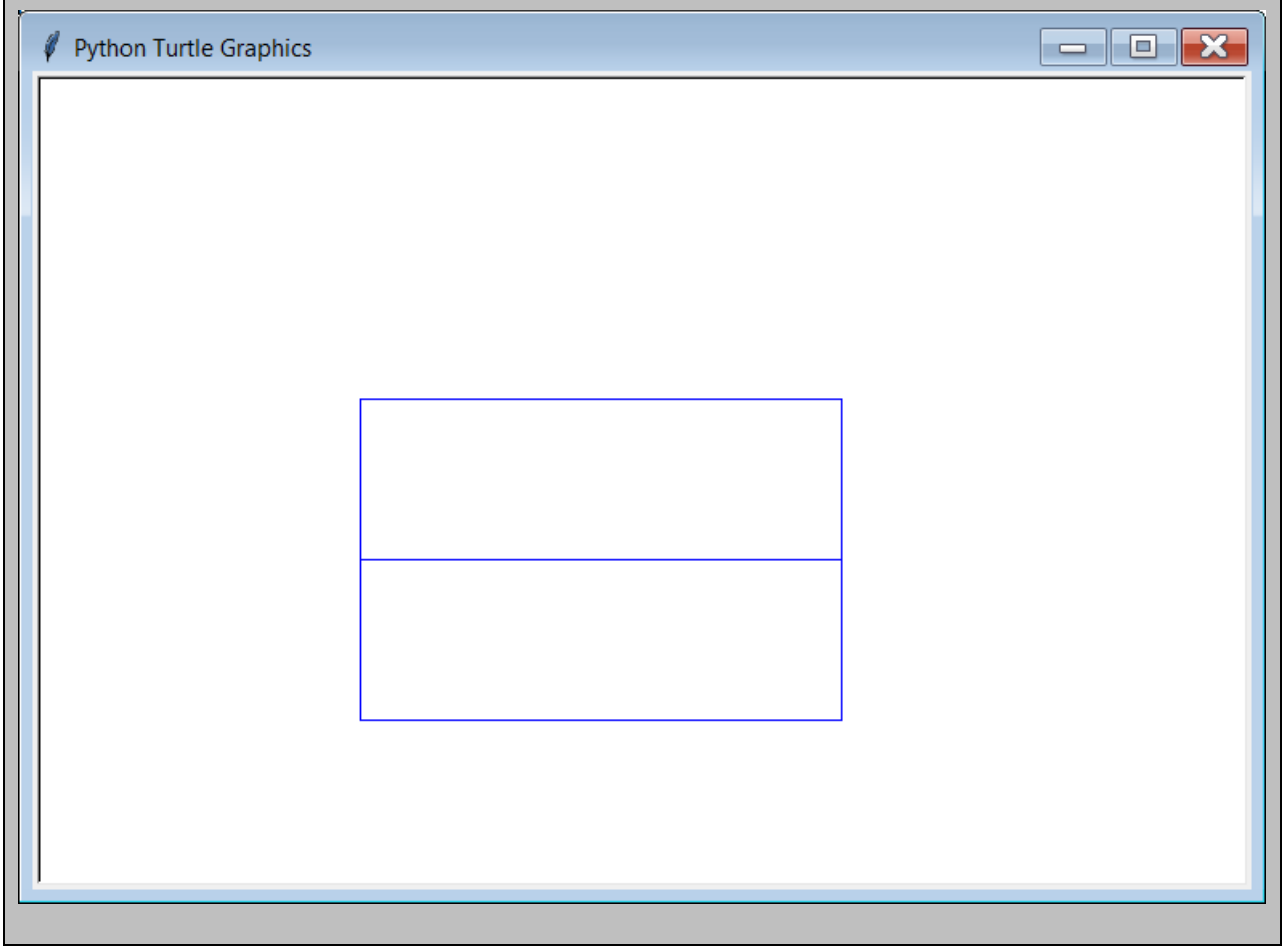
Step 3 – Write the first procedure and make sure it works.

Now that the stubs are in place, you need to pick one of the procedures and actually write it. Program **BigGraphicsStep03.py**, shown in Figure 9.31 demonstrates this buy *implementing* **drawDoors**. To *implement* a subroutine means to write the s*ubroutine body* which refers to the indented program statements. When you finish this first procedure, remember to execute the program to make sure this procedure works before going on to the next one.

Figure 9.31

```
1  # BigGraphicsStep03.py
2  # Creating a Big Graphics Program
3  # Step 3 - Write the first procedure, and make sure it works.
4  # NOTE: Make sure you remove <pass> from this procedure.
5
6
7  from Graphics import *
8
9
10 def drawFloors():
11     setColor("blue")
12     drawRectangle(200,200,500,300)
13     drawRectangle(200,300,500,400)
14
15
16 def drawRoof():
17     pass
18
19
20 def drawChimney():
21     pass
22
23
24 def drawDoor():
25     pass
26
27
28 def drawWindows():
29     pass
30
```

```
31
32
33 ##########
34 #  MAIN  #
35 ##########
36
37 beginGrfx(750,500)
38
39 drawFloors()
40 drawRoof()
41 drawChimney()
42 drawDoor()
43 drawWindows()
44
45 endGrfx()
46
```



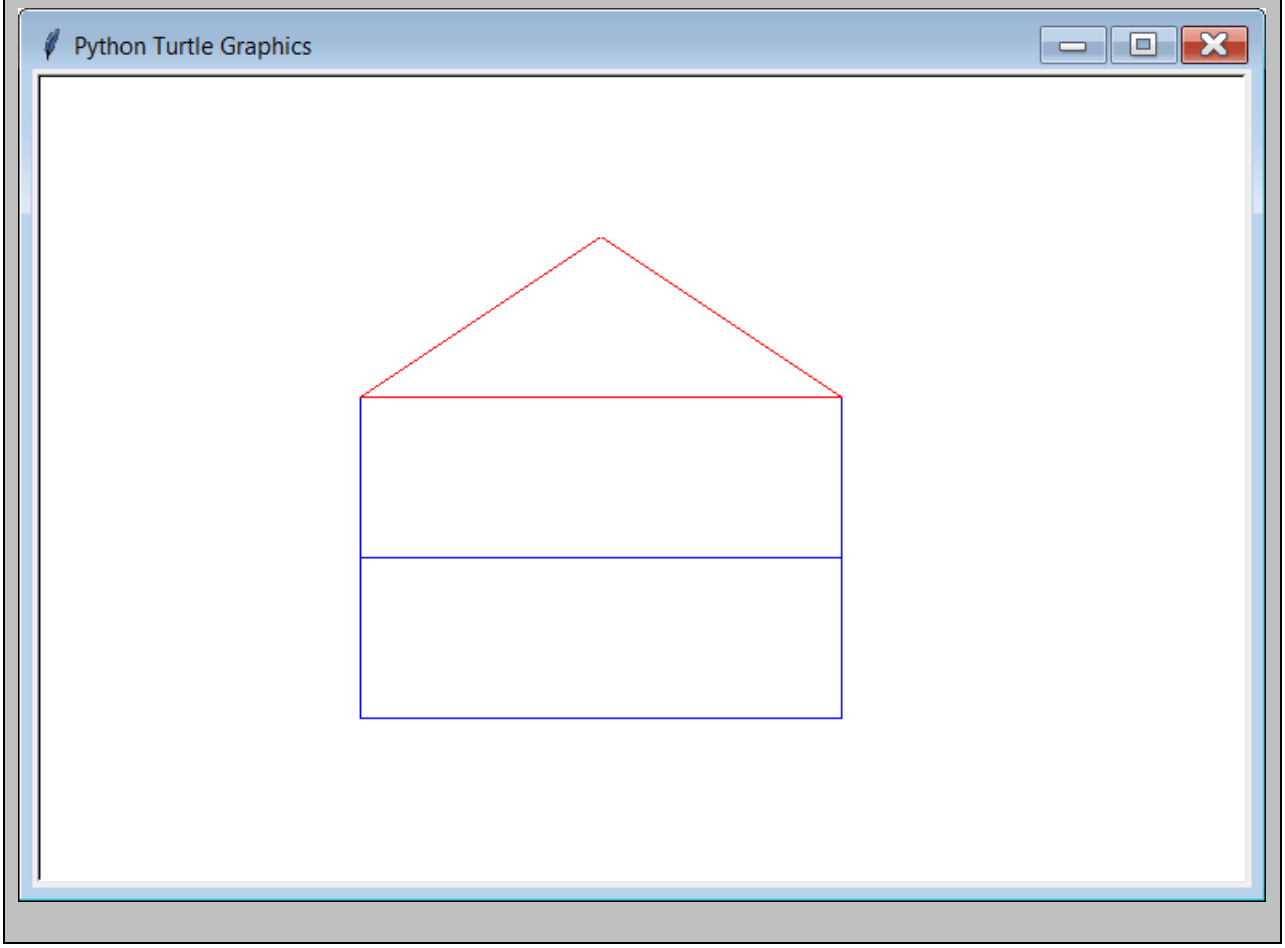
Step 4 – Write the next procedure and make sure it works.

With the first procedure good to go we now move on to the next procedure. Program **BigGraphicsStep04.py**, in Figure 9.32 demonstrates this by implementing **drawRoof**.

Figure 9.32

```
# BigGraphicsStep04.py
# Creating a Big Graphics Program
# Step 4 - Write the next procedure, and make sure it works.
# NOTE: Make sure you remove <pass> from this procedure.


from Graphics import *


def drawFloors():
    setColor("blue")
    drawRectangle(200,200,500,300)
    drawRectangle(200,300,500,400)


def drawRoof():
    setColor("red")
    drawLine(200,200,350,100)
    drawLine(500,200,350,100)
    drawLine(200,200,500,200)


def drawChimney():
    pass


def drawDoor():
    pass


def drawWindows():
    pass

```

```
34
35
36 ##########
37 #  MAIN  #
38 ##########
39
40 beginGrfx(750,500)
41
42 drawFloors()
43 drawRoof()
44 drawChimney()
45 drawDoor()
46 drawWindows()
47
48 endGrfx()
49
```



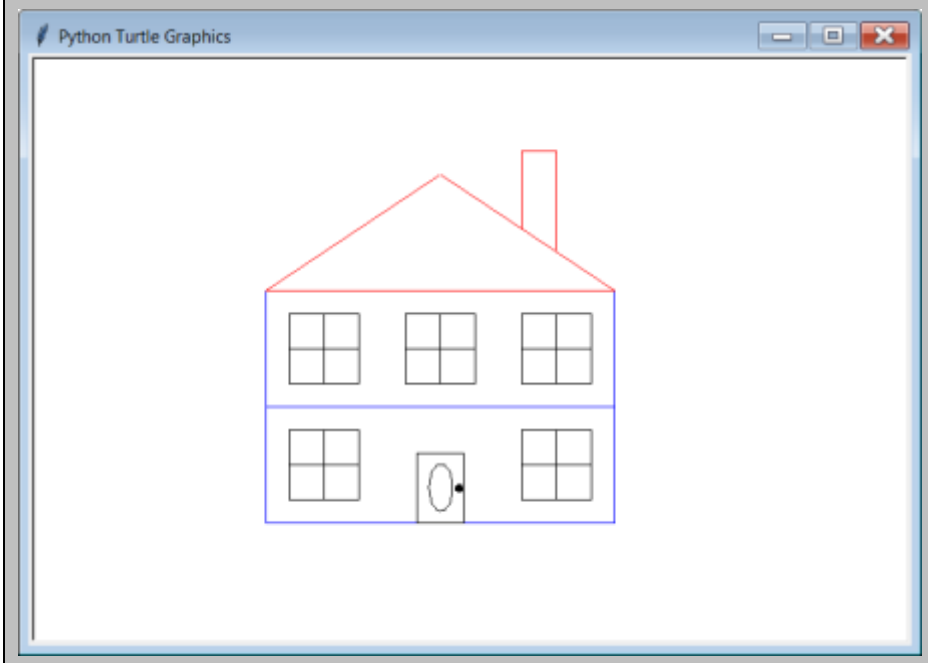
Step 5 – Repeat step 4 until the entire program is done.

After the first couple procedure are done, you just keep repeating the same process by implementing one proxcedureat a time and testing it before you go on to the next one. Program **BigGraphicsStep05.py**, in Figure 9.33 demonstrates the finished program after the rest of the procedures are implemented.

Figure 9.33

```
# BigGraphicsStep05.py
# Creating a Big Graphics Program
# Step 5 - Repeat step 4 until the entire program is done.
# Remember remove <pass> from each procedure as you go.


from Graphics import *


def drawFloors():
    setColor("blue")
    drawRectangle(200,200,500,300)
    drawRectangle(200,300,500,400)


def drawRoof():
    setColor("red")
    drawLine(200,200,350,100)
    drawLine(500,200,350,100)
    drawLine(200,200,500,200)


def drawChimney():
    drawLine(420,146,420,80)
    drawLine(420,80,450,80)
    drawLine(450,80,450,166)


def drawDoor():
    setColor("black")
    drawRectangle(330,340,370,400)
    drawOval(350,370,10,20)
    fillCircle(366,370,3)


def drawWindows():
    drawRectangle(220,220,280,280)
    drawLine(220,250,280,250)
    drawLine(250,220,250,280)
    drawRectangle(420,220,480,280)
    drawLine(420,250,480,250)
```

```
    drawLine(450,220,450,280)
    drawRectangle(320,220,380,280)
    drawLine(320,250,380,250)
    drawLine(350,220,350,280)
    drawRectangle(220,320,280,380)
    drawLine(220,350,280,350)
    drawLine(250,320,250,380)
    drawRectangle(420,320,480,380)
    drawLine(420,350,480,350)
    drawLine(450,320,450,380)



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

beginGrfx(750,500)

drawFloors()
drawRoof()
drawChimney()
drawDoor()
drawWindows()

endGrfx()
```



NOTE: Technically, these 5 steps are not just for graphics programs. They can actually be used to help you write any program, whether it has graphics or not.

9.7 Procedures with a Single Argument & Parameter

You have seen many program statements that call a wide variety of different procedures and functions. The majority of those subroutines use *arguments* to perform a desired computation. You do know that some subroutines do not require arguments, but most do require some type of information for processing.

Subroutine Calls With and Without Arguments

Examples of Subroutine Calls With Arguments:

```
result1 = sqrt(100)
result2 = pow(2,5)
result3 = max(result1,result2)

setColor("red")
drawCircle(650,350,200)
delay(3000)
fillRectangle(100,200,900,600)
```

Examples of Subroutine Calls Without Arguments:

```
displayName()
displayStreetAddress()
displayCityStateZip()

drawFloors()
drawRoof()
drawChimney()
drawDoor()
drawWindows()
```

Why is it that so many subroutines require arguments? This is very natural because subroutines perform some type of task. In most cases the task requires the processing of some type of information. An argument passes this information to a subroutine.

Program **ParameterProcedures01.py**, in Figure 9.34, is a small program with a little procedure named **displayNumner**. Unlike the earlier examples of user-defined procedures in this chapter, the **displayNumber** procedure has a variable (**num**) in its heading. Note that there is also a value (**100**) between the paratheses in the procedure call. When this program executes, and the **displayNumber** procedure is called, the value of **100** gets *passed* from the procedure call to the **num** variable in the procedure heading. To prove that this transfer of information did infact occur, the value of **num** is printed at the end of **displayNumber**. When you look at the output, you see that the number **100** was successfully displayed.

The key difference between this procedure and the procedures from earlier program examples is the *parameter* declaration. All subroutine definitions have an identifier followed by parentheses. If no information is required for the subroutine, the parentheses stay empty. If information is required then the subroutine heading has one or more variables or *parameters* inside the parentheses.

Figure 9.34

```
 1 # ParameterProcedures01.py
 2 # This program passes the argument 100 to the
 3 # parameter <num> in procedure <displayNumber>
 4 # and then displays it.
 5
 6
 7 def displayNumber(num): # Procedure Heading
 8     print()
 9     print("The number is",num)
10
11
12 ###########
13 #  MAIN   #
14 ###########
15
16 displayNumber(100)       # Procedure Call
17
```

```
 ----jGRASP exec: python ParameterProcedures01.py

The number is 100

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

Before moving on we need to get used to some vocabulary. We have been talking about *arguments* since Chapter VI. Starting with this chapter, we will also be talking about *parameters*. Some people use the terms "parameter" and "argument" interchangeably; however, in Python they do have different meanings.

Auguments and *parameters* work together to pass information to a subroutine. The *arguments* are all the values between the parentheses in the subroutine call. The *parameters* are all the variables between the parentheses in the subroutine heading. When you call a subroutine, a copy of each *argument* is passed to its corresponding *parameter*.

Arguments vs. Parameters

Subroutine Heading Example

```
def displayNumber(num):
```

Subroutine Call Example

```
displayNumber(100)
```

The *arguments* are all the values between the parentheses in the subroutine call.

Example: **`100`**

The *parameters* are all the variables between the parentheses in the subroutine heading.

Example: **`num`**

When you call a subroutine, a copy of each *argument* is passed to its corresponding *parameter*.

The Football Analogy

In American football, the quarterback *passes* the ball to the receiver. When working with arguments and parameters the same thing essentially happens. The *argument* is the quarterback. The *parameter* is the receiver. The *data* that is being passed from the *argument* to the *parameter* is the football.

Arguments/Parameters Disclaimer

If you learned another programming language before taking this course. You may have learned about *Actual Parameters* in the subroutine call and *Formal Parameters* in the subroutine heading. For the sake of this course, and to be consistent with the terminology used on the PCEP Certification Exam, just do the following translation in your head:

Other Languages		Python
Actual Parameters	→	Arguments
Formal Parameters	→	Parameters

The parameters (the variables in the subroutine heading) indicate the number of required arguments (the values in the subroutine call). The previous program example used a constant integer to pass the information. There are many ways to pass information and program **ParameterProcedures02.py**, in Figure 9.35, shows that the argument can take four different formats. It is possible to use a constant only, a variable only, an expression with constants and/or variables, and a functions call.

Figure 9.35

```
 1 # ParameterProcedures02.py
 2 # This program demonstrates that an argument
 3 # can be: a constant, like <100> or <pi>,
 4 # a variable, like <x>, an expression with
 5 # constants and/or variables, like <20 + 30> or
 6 # <4 * x>, and a function call like <sqrt(225)>.
 7
 8
 9 from math import *
10
11
12 def displayNumber(num):
13     print()
14     print("The number is",num)
15
16
17
18 ###########
19 #  MAIN   #
20 ###########
21
22 x = 200
23 displayNumber(100)
24 displayNumber(x)
25 displayNumber(20 + 30)
26 displayNumber(4 * x)
27 displayNumber(pi)
28 displayNumber(sqrt(225))
```

```
 ----jGRASP exec: python ParameterProcedures02.py

The number is 100

The number is 200

The number is 50

The number is 800

The number is 3.141592653589793

The number is 15.0

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

Argument Formats

Arguments can be:	Example:
• constants	**100** or **pi**
• variables	**x**
• expressions with constants only	**20 + 30**
• expressions with variables & constants	**x + 5**
• function calls	**sqrt(225)**

9.8 Procedures with Multiple Arguments & Parameters

You just saw that arguments can take on many different formats, but what about quantity. Are procedures limited to a single argument? Now you may be new to creating your own procedures, but you have used procedure calls like **drawLine(x1,y1,x2,y2)**. The **drawLine** procedure has four arguments. Using more than one argument is certainly common in Python and it is not very difficult, but there are special considerations. Program **ParameterProcedures03.py**, in Figure 9.36, demonstrates a procedure that computes and displays the area of a rectangle.

Note that the **rectangleArea** procedure is called twice with the same values, but the second time the order of the arguments is reversed. The output is identical and may give the impression that argument sequence is not an issue. The program is misleading. The only reason the argument sequence is not an issue in this particular example is because we are multiplying and multiplication is *commutative*. As you will see, 99% of the time, argument sequence is very important.

Figure 9.36

```
1 # ParameterProcedures03.py
2 # This program demonstrates passing two arguments to a
3 # procedure.  Procedure <showRectangleArea> is called twice.
4 # In this case reversing the sequence of the arguments
5 # is not a problem.
6
7
8 def showRectangleArea(L,W):
9     area = L * W
10    print()
11    print("The rectangle area is",area)
12
13
14
15 ##########
16 #  MAIN  #
17 ##########
18
19 length = 100
20 width = 50
21 showRectangleArea(length,width)
22 showRectangleArea(width,length)
23
```

```
----jGRASP exec: python ParameterProcedures03.py

The rectangle area is 5000

The rectangle area is 5000

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

The reason program **ParameterProcedures03.py** has no problem with sequence is due to the fact that the procedure operation is multiplication. You should know from Algebra that **P * Q** equals **Q * P**. Mathematical operations do not all have the same properties. How about subtraction? Can you say that **P – Q** = **Q – P**?

This very question is answered by program **ParameterProcedures04.py**, in Figure 9.37. Once again there is a user-defined procedure that displays the result of a mathematical operation, but this time it computes the difference. Procedure **showDifference** computes **a – b** and in this case the sequence of the calling arguments is very significant. The two outputs are now very different.

Figure 9.37

```
1  # ParameterProcedures04.py
2  # This program demonstrates that argument sequence
3  # matters. In this example procedure <showDifference>
4  # will display different results when the calling
5  # arguments are reversed.
6
7
8  def showDifference(a,b):
9     difference = a - b
10    print()
11    print("The difference is",difference)
12
13
14
15 ##########
16 #  MAIN  #
17 ##########
18
19 num1 = 100
20 num2 = 50
21 showDifference(num1,num2)
22 showDifference(num2,num1)
23
```

```
  ----jGRASP exec: python ParameterProcedures04.py

The difference is 50

The difference is -50

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

Argument Sequence Matters

The first argument passes information to the first parameter.

The second argument passes information to the second parameter.

Arguments placed out of sequence may result in *syntax errors*, *run-time errors* or *logic errors*.

Something that is just as important as making sure your data is in the correct sequence is also making sure your data is the correct type. Program **ParameterProcedures05.py**, in Figure 9.38, uses the same **displayDifference** procedure from the previous program. The difference now (no pun intended) is that the program attempts to pass string values to the **displayDifference** procedure. Since *subtraction* is something that only works with numbers, this is not going to work. The error message tells you that string values are **unsupported operands** for the minus sign (–).

Figure 9.38

```
1  # ParameterProcedures05.py
2  # This program demonstrates that argument data types also
3  # matter. In this example 2 string arguments were passed
4  # to procedure <showDifference>.  This does not work as
5  # "subtraction" is something that only works with numbers.
6
7
8  def showDifference(a,b):
9     difference = a - b
10    print()
11    print("The difference is",difference)
12
13
14
15 ##########
16 #  MAIN  #
17 ##########
18
19 num1 = "John"
```

```
20 num2 = "Smith"
21 showDifference(num1,num2)
22 showDifference(num2,num1)
23
```

```
  ----jGRASP exec: python ParameterProcedures05.py
  Traceback (most recent call last):
  File "ParameterProcedures05.py", line 21, in
<module>
      showDifference(num1,num2)
    File "ParameterProcedures05.py", line 9, in
showDifference
      difference = a - b
  TypeError: unsupported operand type(s) for -:
'str' and 'str'

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

You have seen several programs using subroutines with multiple parameters. In each case the multiple parameters were all the same data type. This is actually not a requirement. Program **ParameterProcedures06.py**, in Figure 9.39, demonstrates a *multi-type* procedure using four different parameter data types. The **showStudentInfo** procedure has 4 parameters which are **name**, **age**, **gpa** and **inState**. It is expected that the arguments will be a string, integer, real number and a boolean value, in that sequence.

The program actually calls the **showStudentInfo** procedure 3 times, each with 3 different sets of data. The first 2 times the procedure is called, the arguments are in the proper order; however, with the third procedure call, the arguments are completely out of order. In this particular case, there is no syntax error, but the program output would definitely indicate that something is wrong.

Figure 9.39

```
1  # ParameterProcedures06.py
2  # This program demonstrates that different data types
3  # can be passed to the same procedure.
4  # The <showStudentInfo> procedure is called 3 times.
5  # The first two procedure calls are proper.
6  # The third one has the arguments out of order
7  # which causes strange output.
8
9
10 def showStudentInfo(name, age, gpa, inState):
11     print()
12     print("Student Information:")
13     print("Name:     ",name)
14     print("Age:      ",age)
15     print("GPA:      ",gpa)
16     print("In-State: ",inState)
17
18
19
20 ##########
21 #  MAIN  #
22 ##########
23
24 showStudentInfo("John Smith", 22, 2.875, True)
25 showStudentInfo("Suzy Brown", 29, 3.999, False)
26 showStudentInfo(1.763, True, "Tom Jones", 27)
27
```

```
  ----jGRASP exec: python ParameterProcedures06.py

Student Information:
Name:       John Smith
Age:        22
GPA:        2.875
In-State:   True
```

```
Student Information:
Name:       Suzy Brown
Age:        29
GPA:        3.999
In-State:   False

Student Information:
Name:       1.763
Age:        True
GPA:        Tom Jones
In-State:   27

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

This section will finish with a track analogy. Look at the diagrams carefully and you will find that they illustrate the correct way to use arguments and parameters. As far as programming is concerned, writing procedures with parameters will simplify your life. Your programs will be better designed. Your programs will be much shorter by using the same subroutine for multiple situations. At the same time, your programs can give you headaches if arguments and/or parameters are not used correctly. Remember that *arguments* and *parameters* must match in 3 different ways: quantity, type and sequence.

The Track Relay Analogy

Let us summarize this argument and parameter business, with a real life analogy that may help some students. The analogies that follow explain some of the rules for arguments and parameters in a totally different manner. Imagine that you are at a track meet and you are watching a relay race. In this relay race the starters run 100 meters and then pass a baton to the next runner in their team.

In the first relay race example, the race official checks the track and announces that the race is not ready. A look at **Race-1** shows there are four starters ready in their lanes, but only three runners at the 100 meter *baton passing* mark. A runner from the Netherlands (NL) is missing. If the four starting runners represent *arguments* and the three runners at the 100 meter mark represent *parameters* we have a situation where the *quantities* of arguments and parameters do not match.

Race-1

US	US
GB	GB
FR	FR
NL	

Race-2 presents another situation with a different problem. This time the number of runners is correct. There are four starters and there are also four runners at the 100 meter mark ready to receive a baton. However two runners at the 100 meter mark are standing in the wrong lanes. The track official announces that the race cannot start unless the runners change lanes and are ready to receive the batons from their own countrymen. This represents a situation where the *sequence* of arguments and parameters do not match.

Race-2

US	US
GB	GB
FR	NL
NL	FR

Race3 is not a problem situation. This race demonstrates an analogy to help explain the naming of arguments and parameters. In **Race3**, runner *John* starts for the United States (US) and passes to *Greg*. *George* starts for Great Britain (GB) and passes to *Thomas*. *Gerard* starts for France (FR) and passes to *Louis*. Finally, *Hans* starts for the Netherlands and passes to another guy named *Hans*.

Race-3

US (John)	US (Greg)
GB (George)	GB (Thomas)
FR (Gerald)	FR (Louis)
NL (Hans)	NL (Hans)

The point of this analogy is that the names do not matter. They can be the same. They can be different. It works either way. What does matter is that there are the same number of runners at the passing mark as there are in the starting blocks. It also matters that everybody stays in their lanes and that the runners receiving batons are on the same team as the starters.

Note: The batons are not passed based on the names of the runners. They are passed based on the lanes in which they run.

Important Rules About Using Subroutines With Parameters

The number of *arguments* (values in the subroutine call) must match the number of *parameters* (variables in the subroutine heading).

The corresponding arguments must match the intended data type of the parameters.

The sequence of the arguments must match the sequence of the parameters.

The identifiers of the arguments may be the same as or may be different from the identifiers of the parameters.

9.9 Functions with a Single Argument & Parameter

Every user-defined subroutine that you have seen thus far in this chapter has been a *procedure*. Even before you started to create your own procedures, you were using procedures like **drawCircle** and **fillRectangle** that were created by someone else. But even before that, you were using *functions* of the **math** library like **sqrt** and **abs**. *Functions* and *procedures* are very similar. Both are *subroutines*. The only difference is what when a function is finished, it *returns* a value. When a procedure is finished, it does not return anything. It just ends.

Program **Functions01.py**, in Figure 9.40, shows the user-created function **getNextNumber**. Notice that the last statement in the function body is a **return** statement. Please realize that all functions require a **return** statement, and it will usually be the last statement in the function. The fact that a subroutine *returns* something is what makes it a *function*.

Figure 9.40

```
 1 # Functions01.py
 2 # This program introduces a "function" with one parameter.
 3 # Function <getNextNumber> returns the next integer after
 4 # the value passed to its parameter.
 5 # NOTE: A function is a subroutine that returns a value.
 6 # A procedure is a subroutine that does not return a value.
 7
 8
 9 from random import randint
10
11
12 def getNextNumber(current):
13     next = current + 1
14     return next
15
16
17
18 ##########
19 #  MAIN  #
20 ##########
21
22 for k in range(10):
23     randNum = randint(10,99)
24     print()
25     print("Random number:",randNum)
26     print("Next number:  ",getNextNumber(randNum))
27
```

```
  ----jGRASP exec: python Functions01.py

Random number: 93
Next number:   94

Random number: 36
Next number:   37

Random number: 47
Next number:   48
```

```
Random number: 29
Next number:   30

Random number: 77
Next number:   78

Random number: 20
Next number:   21

Random number: 78
Next number:   79

Random number: 11
Next number:   12

Random number: 58
Next number:   59

Random number: 76
Next number:   77

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

Program **Functions02.py**, in Figure 9.41, shows a practical example of a Function. In this case the user-created subroutine, **checkPIN**, return the boolean value **true** or **false** depending upon the correct entry of a PIN (**P***ersonal* **I***dentification* **N***umber*) that is needed to use a debit or credit card at an ATM (**A***utomatic* **T***eller* **M***achine*). The argument passed is the PIN number. This number is compared to **1234** and if the number matches **True** is returned and **False** otherwise. This function also shows the first practical example of using a **boolean** variable. It is true that Boolean variables only have two possible values, but these two values can be very practical. The real benefit of Boolean variables is they help to make your programs more readable. You will see more Boolean examples in Chapter XI.

Figure 9.41

```
1  # Functions02.py
2  # This example returns a Boolean value, which is used
3  # frequently to check for correct user keyboard input.
4  # NOTE: This program also demonstrates the true purpose of
5  # Boolean variables.  They make the program more readable.
6
7
8  def checkPIN(pin):
9      if pin == 1234:
10        return True
11     else:
12        return False
13
14
15
16 ##########
17 #  MAIN  #
18 ##########
19
20 correctPIN = False
21 while(not correctPIN):
22    pin = eval(input("\nEnter your 4 digit PIN  -->  "))
23    correctPIN = checkPIN(pin)
24    if not correctPIN:
25       print("\nIncorrect PIN.  Please try again.")
26
27 print("\nYou have successfully logged in.")
28 print("Select your bank transaction:")
29
```

```
   ----jGRASP exec: python Functions02.py

▶▶ Enter your 4 digit PIN  -->  9876

   Incorrect PIN.  Please try again.
```

```
Enter your 4 digit PIN  -->  5555

Incorrect PIN.  Please try again.

Enter your 4 digit PIN  -->  1234

You have successfully logged in.
Select your bank transaction:

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

9.10 Functions with Multiple Arguments & Parameters

Functions, like procedures can have multiple parameters. Program **Functions03.py**, in Figure 9.42, serves double duty. First, the program shows an example of a function with two parameters. The function receives two parameters and returns their sum. Second, the program compares the similarities and differences of *procedure*s with *function*s by using both types of subroutines in the same program.

Figure 9.42

```
1 # Functions03.py
2 # This program has 2 "add" subroutines.
3 # <add1> is a procedure.  <add2> is a function.
4 # The purpose is to demonstrate the differences
5 # between these 2, which are:
6 # 1) They are called differently.
7 # 2) Functions end with a <return> command.
8
```

```
 9
10 def add1(n1,n2):
11     sum = n1 + n2
12     print(n1,"+",n2,"=",sum)
13
14
15 def add2(n1,n2):
16     sum = n1 + n2
17     return sum
18
19
20 ##########
21 #  MAIN  #
22 ##########
23
24 num1 = 1000
25 num2 = 100
26 print()
27 add1(num1,num2)
28 print(num1,"+",num2,"=",add2(num1,num2))
29
```

```
  ----jGRASP exec: python Functions03.py


 1000 + 100 = 1100
 1000 + 100 = 1100


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

Procedures and Functions

Procedure Example

```
def add1(n1,n2):
    sum = n1 + n2
    print(n1,"+",n2,"=",sum)
```

Procedures do not return a value.

Function Example

```
def add2(n1,n2):
    sum = n1 + n2
    return sum
```

Functions do return a value. All functions must have a **return** statement, which is usually the last statement in the function.

Program **Functions04.py**, in Figure 9.43, shows a four-function calculator. There are four user-defined Functions for **add**, **subtract**, **multiply** and **divide** operations. Each function is very similar and uses the same two parameters for a binary operation. It is very important to understand that functions are *called* in a program statement that uses the returned value.

Figure 9.42

```
1 # Functions04.py
2 # This program demonstrates a 4 "function" calculator.
3 # NOTE: While it may be good program design to put all
4 # of these functions in a separate library, most of the
5 # examples in the book will continue to put the entire
6 # program in a single file for the same of simplicity.
7
```

```
 8
 9 def add(n1,n2):
10     return n1 + n2
11
12
13 def subtract(n1,n2):
14     return n1 - n2
15
16
17 def multiply(n1,n2):
18     return n1 * n2
19
20
21 def divide(n1,n2):
22     return n1 / n2
23
24
25
26 ###########
27 #  MAIN   #
28 ###########
29
30 num1 = 1000
31 num2 = 100
32 print()
33 print(num1,"+",num2,"=",add(num1,num2))
34 print(num1,"-",num2,"=",subtract(num1,num2))
35 print(num1,"*",num2,"=",multiply(num1,num2))
36 print(num1,"/",num2,"=",divide(num1,num2))
```

```
  ----jGRASP exec: python Functions04.py

1000 + 100 = 1100
1000 - 100 = 900
1000 * 100 = 100000
1000 / 100 = 10.0

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

We have seen that procedures and functions are called differently. Procedures are like complete sentences. They can exist on their own. Functions are like subordinate clauses. They are incomplete and need to be part of something else. This is because a function *returns* a value and you need to do something with that value. You can print it, or store it or compare it to another value. Whatever; you just need to do something with it.

This is all demonstrated in program **Functions05.py**, shown in Figure 9.43. In this program, the **add function** is called in 4 different places. First it is called with an output display using a **print** statement. Then it is called with an assignment statement. Then, it is called in a comparison made by an **if** statement. In each of the first 3 examples you will note the value returned by the **add** function is used in some way in the program statement. If you look at line 28, you will see that the **add** function is actually called a fourth time, on a line by itself. In this case, nothing is done with the returned value. Because of this, the program statement essentially does nothing. Yes it does add **800** and **900**. Yes the value of **1700** is returned, but then what? Nothing is done with the **1700**.

Figure 9.43

```
 1 # Functions05.py
 2 # This program demonstrates 3 proper ways
 3 # and 1 improper way to call a function.
 4
 5
 6 def add(n1,n2):
 7     return n1 + n2
 8
 9
10
11 ###########
12 #  MAIN   #
13 ###########
14
15 print()
16 print("Sum:",add(200,300))
17
18 sum = add(400,500)
19 print("Sum:",sum)
20
```

```
21 checking = 600
22 savings  = 700
23 if add(checking,savings) <= 0:
24    print("We are broke!")
25 else:
26    print("Let's go shopping!")
27
28 add(800,900)  # Essentially does nothing
29
```

```
  ----jGRASP exec: python Functions05.py

Sum: 500
Sum: 900
Let's go shopping!

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

9.11 Creating Subroutines from Other Subroutines

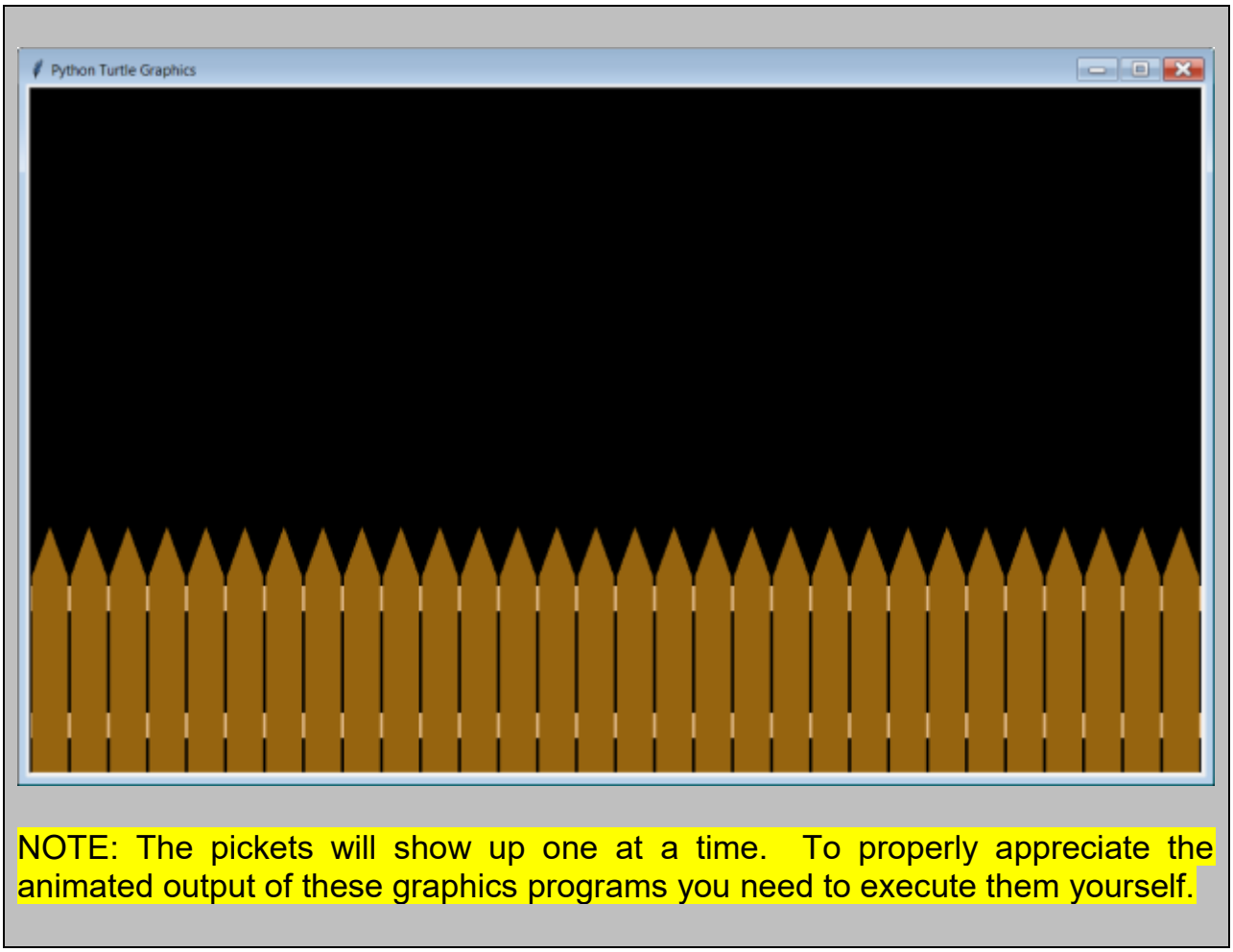
Consider the *Tool* analogy from the introduction to this chapter. Essentially, we make tools so that we can make the stuff we want. What if we use some tools to make a bigger tool, like a crane? Now we can make bigger stuff, like a car, a house or even a building. This exact same concept applies to programming. Once subroutines are created, they can be used to create bigger, more powerful subroutines.

In this section, we will create some user-defined subroutines and then we will use those newly created subroutines to create more complex subroutines. The first program starts with a modest little procedure that displays a single brown picket.

The **picket** procedure in **SubFromSub01.py**, shown in Figure 9.44, has only two program statements. It is small, but the program will steadily grow to an interesting graphics display with many user-defined subroutines.

Figure 9.44

```
# SubFromSub01.py
# This program demonstrates a <picket> procedure
# that will be used to help draw a fence.



from Graphics import *


def drawPicket(x):
    fillPolygon([x,700,x,500,x+18,450,x+36,500,x+36,700])
    delay(250)  # delay for 1/4 of a second



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

beginGrfx(1200,700)

setBackground("black")
setColor("burlywood")
fillRectangle(0,510,1200,535)
fillRectangle(0,640,1200,665)

setColor("brown")
for x in range(2,1200,40):
    drawPicket(x)

endGrfx()

```



NOTE: The pickets will show up one at a time. To properly appreciate the animated output of these graphics programs you need to execute them yourself.

The next program example, **SubFromSub02.py**, in Figure 9.45, displays the same picket fence. This time we have an example of a user-defined procedure using another user-defined procedure. There is a new **fence** procedure, which uses the previously created **picket** procedure. This follows the philosophy of not *reinventing the wheel*.

Figure 9.45

```
1 # SubFromSub02.py
2 # This program uses the <drawPicket> procedure
3 # to create the <drawFence> procedure.
4
5
6 from Graphics import *
7
8
```

```
 9 def drawPicket(x):
10    fillPolygon([x,700,x,500,x+18,450,x+36,500,x+36,700])
11    delay(250)  # delay for 1/4 of a second
12
13
14 def drawFence():
15    # cross beams
16    setColor("burlywood")
17    fillRectangle(0,510,1200,535)
18    fillRectangle(0,640,1200,665)
19    # pickets
20    setColor("brown")
21    for x in range(2,1200,40):
22       drawPicket(x)
23
24
25 ##########
26 #  MAIN  #
27 ##########
28
29 beginGrfx(1200,700)
30
31 setBackground("black")
32 drawFence()
33
34 endGrfx()
35
```

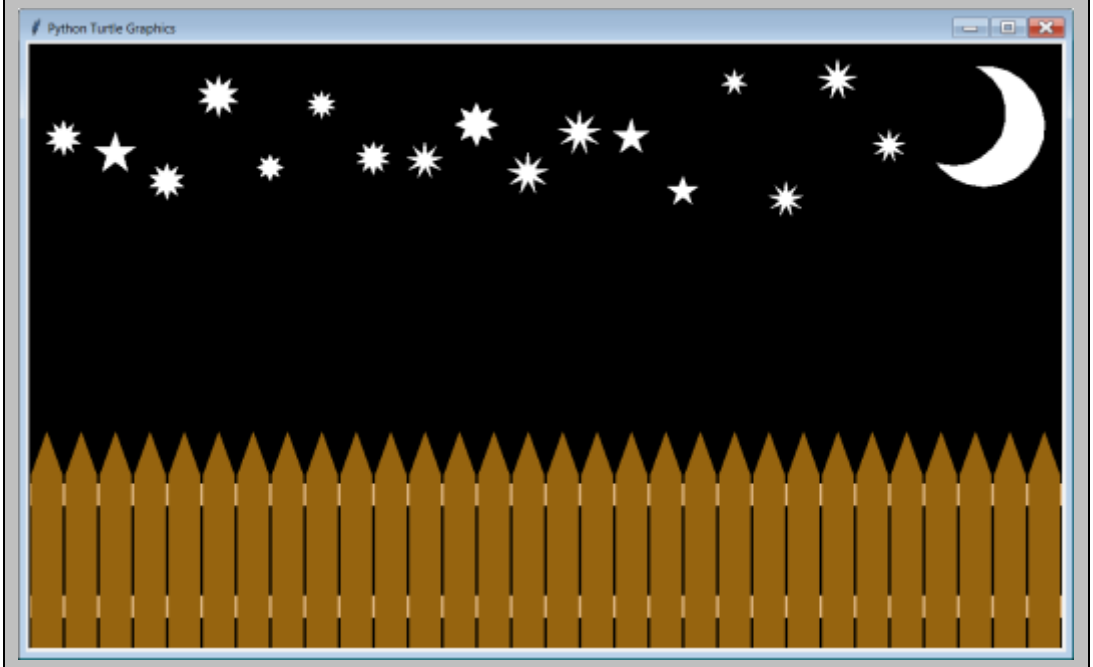
The output is identical to the previous program.

You may have noticed that the previous 2 programs started by making the entire applet **black**. The black background is intentional. We are looking at a picket fence at night. There are a couple things missing from our night sky, namely the moon and some stars. Program **SubFromSub03.py**, in Figure 9.46, fixes this with a new **nightSky** procedure which starts by making the background **black** and then calls both the **drawMoon** and **drawRandomStar** procedures. As before, the **fence** procedure calls the **picket** procedure.

Figure 9.46

```
# SubFromSub03.py
# This program adds procedure <drawNightSky>,
# which is created using procedures <drawMoon>
# and <drawRandomStar>.



from Graphics import *
from random import randint


def drawPicket(x):
   fillPolygon([x,700,x,500,x+18,450,x+36,500,x+36,700])
   delay(250)  # delay for 1/4 of a second


def drawFence():
   # cross beams
   setColor("burlywood")
   fillRectangle(0,510,1200,535)
   fillRectangle(0,640,1200,665)
   # pickets
   setColor("brown")
   for x in range(2,1200,40):
      drawPicket(x)


def drawMoon():
   setColor("white")
   fillCircle(1110,95,70)
   setColor("black")
   fillCircle(1075,80,60)


def drawRandomStar(x):
   y = randint(40,200)
   radius = randint(15,25)
   points = randint(5,10)
   fillStar(x,y,radius,points)
   delay(250)

```

```
42
43 def drawNightSky():
44     setBackground("black")
45     drawMoon()
46     setColor("white")
47     for x in range(40,1001,60):
48         drawRandomStar(x)
49
50
51
52 ##########
53 #  MAIN  #
54 ##########
55
56 beginGrfx(1200,700)
57
58 drawNightSky()
59 drawFence()
60
61 endGrfx()
62
```



Program **SubFromSub04.py**, in Figure 9.47, goes back to the "House" and "Tree" that were drawn back in program **GraphicsProcedures08.py**. This time, the "**MAIN**" section of the program has 3 procedure calls instead of 9. This is because **drawTrunk** and **drawLeaves** are now called by the new **drawTree** procedure; **drawSky** and **drawGrass** are now called by the new **drawBackground** procedure; and all of the procedures tat drew parts of the house, are now called by the new **drawHouse** procedure. It was mentioned that in a well organized program, the "**MAIN**" section should look like an outline. In this program example, we essentially have a multi-level outline.

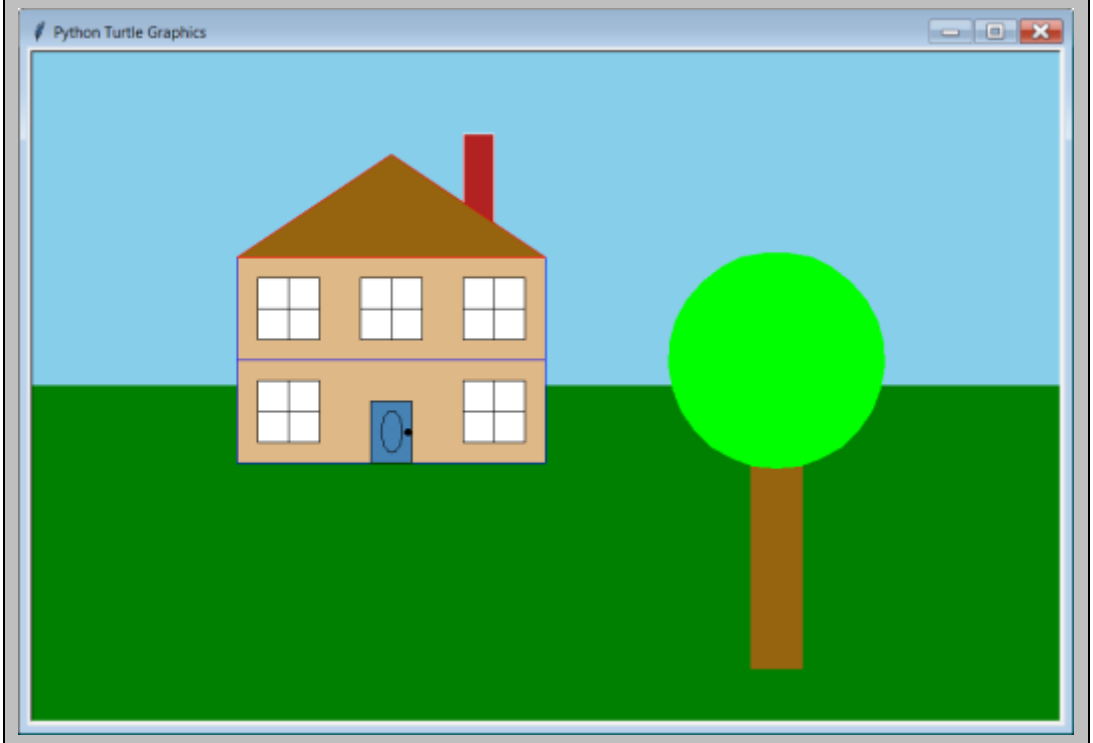
Figure 9.47

```
 1 # SubFromSub04.py
 2 # This program program revisits the "House"
 3 # from program GraphicsProcedures08.py
 4 # Now instead of 9 separate procedure calls,
 5 # we have 3: <drawBackground>, <drawHouse>
 6 # and <drawTree>.  In a sense, the program
 7 # now resembles a multi-level outline.
 8
 9
10 from Graphics import *
11
12
13 # House procedures
14
15 def drawFloors():
16     setColor("burlywood")
17     fillRectangle(200,200,500,400)
18     setColor("blue")
19     drawRectangle(200,200,500,300)
20     drawRectangle(200,300,500,400)
21
22
23 def drawRoof():
24     setColor("brown")
25     fillPolygon([200,200,350,100,500,200])
26     setColor("red")
27     drawLine(200,200,350,100)
28     drawLine(500,200,350,100)
29     drawLine(200,200,500,200)
30
31
```

```
32 def drawChimney():
33     setColor("firebrick")
34     fillPolygon([420,146,420,80,450,80,450,166])
35     setColor("white")
36     drawLine(420,146,420,80)
37     drawLine(420,80,450,80)
38     drawLine(450,80,450,166)
39 
40 
41 def drawDoor():
42     setColor("steel blue")
43     fillRectangle(330,340,370,400)
44     setColor("black")
45     drawRectangle(330,340,370,400)
46     drawOval(350,370,10,20)
47     fillCircle(366,370,3)
48 
49 
50 def drawWindows():
51     setColor("white")
52     fillRectangle(220,220,280,280)
53     fillRectangle(420,220,480,280)
54     fillRectangle(320,220,380,280)
55     fillRectangle(220,320,280,380)
56     fillRectangle(420,320,480,380)
57     setColor("black")
58     drawRectangle(220,220,280,280)
59     drawLine(220,250,280,250)
60     drawLine(250,220,250,280)
61     drawRectangle(420,220,480,280)
62     drawLine(420,250,480,250)
63     drawLine(450,220,450,280)
64     drawRectangle(320,220,380,280)
65     drawLine(320,250,380,250)
66     drawLine(350,220,350,280)
67     drawRectangle(220,320,280,380)
68     drawLine(220,350,280,350)
69     drawLine(250,320,250,380)
70     drawRectangle(420,320,480,380)
71     drawLine(420,350,480,350)
72     drawLine(450,320,450,380)
73 
74 
```

```
# Tree procedures

def drawTrunk():
    setColor("brown")
    fillRectangle(700,400,750,600)


def drawLeaves():
    setColor("green")
    fillCircle(725,300,105)


# Background procedures

def drawSky():
    setColor("sky blue")
    fillRectangle(0,0,1000,325)


def drawGrass():
    setColor("dark green")
    fillRectangle(0,325,1000,650)


# Procedures made from other procedures

def drawBackground():
    drawSky()
    drawGrass()


def drawHouse():
    drawFloors()
    drawRoof()
    drawChimney()
    drawDoor()
    drawWindows()


def drawTree():
    drawTrunk()
    drawLeaves()
```

```
117
118
119
120 ##########
121 #  MAIN  #
122 ##########
123
124 beginGrfx(1000,650)
125
126 drawBackground()
127 drawHouse()
128 drawTree()
129
130 endGrfx()
```



In spite of what you have seen in the previous 4 program examples, this section is not called "Creating Procedures from Other Procedures". It is called "Creating Subroutines from Other Subroutines". Why am I using the more general term here? The reason is that in the same way that you can create a procedure from another procedure, you can also create a function from another function.

Program **SubFromSub05.py**, in Figure 9.48, demonstrate 2 user-created mathematical functions. These are **gcf** (for "Greatest Common Factor") and **lcm** (for "Least Common Multiple"). If you look at the code for the **gcf** function it seems a little complex. Do not worry about this. Another benefit of using subroutines is that you can use a subroutine even if you do not know how it works. You just need to know what it does and what parameters (if any) to pass. Consider this. Do you need to know how to build a car in order to drive a car? No, you do not. The same thing applies here.

OK, back to the program now. It helps if you know that the least common multiple of 2 numbers is equal to the product of those 2 numbers divided by their greatest common factor. With that in mind, along with a provided **gcf** function, writing the **lcm** function should not be that complicated. On line 25 we see the entire body of the **lcm** function is a single **return** statement. Note that this return statement calls the **gcf** function.

Figure 9.47

```
 1 # SubFromSub05.py
 2 # This program demonstrates that a function can also be
 3 # created from another function. In this case, the <gcf>
 4 # function is used to create the <lcm> function.
 5 # Example: LCM(A,B) = A * B / GCF(A,B)
 6 # This program also demonstrates that you can call a
 7 # function if you know what it does and what parameters
 8 # it requires, but you do not necessarily need to know
 9 # how it works. This program also demonstrates how you
10 # can break up a very long command and make it "wrap"
11 # to the next line by using a backslash (\).
12
13
14 def gcf(a,b):
15     while True:
16         rem = a % b
17         if rem == 0:
18             return b
19         else:
20             a,b = b,rem
21
22
23 def lcm(a,b):
24     return a * b // gcf(a,b)
25
```

```
26
27
28 ##########
29 #  MAIN  #
30 ##########
31
32 print()
33 num1 = eval(input("Enter 1st number  -->  "))
34 num2 = eval(input("Enter 2nd number  -->  "))
35 print()
36 print("The Greatest Common Factor of",num1, \
37 "and",num2,"is",gcf(num1,num2))
38 print()
39 print("The Least Common Multiple of",num1, \
40 "and",num2,"is",lcm(num1,num2))
41
```

```
  ----jGRASP exec: python SubFromSub05.py

▶▶ Enter 1st number  -->  4000
▶▶ Enter 2nd number  -->  625

   The Greatest Common Factor of 4000 and 625 is 125

   The Least Common Multiple of 4000 and 625 is 20000

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

There is something else that this program demonstrates. If you have a really long program statement, you can split it into 2 separate program statements by using a backslash (\ ). Keep in mind that this does not work for long string literals.