Recursion

3

Don't think too hard! That's one of the central themes of this chapter. It's not often that you tell computer programmers not to think too hard, but this is one time when it is appropriate. You need to read this chapter if you have not written recursive functions before. Most computer science students start by learning to program in a style called *imperative* programming. This simply means that you are likely used to thinking about creating variables, storing values, and updating those values as a program proceeds. In this chapter you are going to begin learning a different style of programming called *functional* programming. When you program in the functional style, you think much more about the definition of *what* you are programming than *how* you are going to program it. Some say that writing recursive functions is a *declarative* approach rather than an *imperative* approach. You'll start to learn what that means for you very soon. When you start to get good at writing recursive functions you'll be surprised how easy it can be!

Python programs are executed by an interpreter. An interpreter is a program that reads another program as its input and does what it says. The Python interpreter, usually called *python*, was written in a language called C. That C program reads a Python program and does what the Python program says to do in its statements. An interpreter interprets a program by running or executing what is written within it. The interpreter interacts with the operating system of the computer to use the network, the keyboard, the mouse, the monitor, the hard drive, and any other I/O device that it needs to complete the work that is described in the program it is interpreting. The picture in Fig. 3.1 shows you how all these pieces fit together.

In this chapter we'll introduce you to scope, the run-time stack, and the heap so you understand how the interpreter calls functions and where local variables are stored. Then we'll provide several examples of recursive functions so you can begin to see how they are written. There will be a number of recursive functions for you to practice writing and we'll apply recursion to drawing pictures as well.

One thing you will not do in the homework for this chapter is write code that uses a *for* loop or a *while* loop. If you find yourself trying to write code that uses either kind of loop you are trying to write a function imperatively rather than functionally.

K.D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-13072-9_3

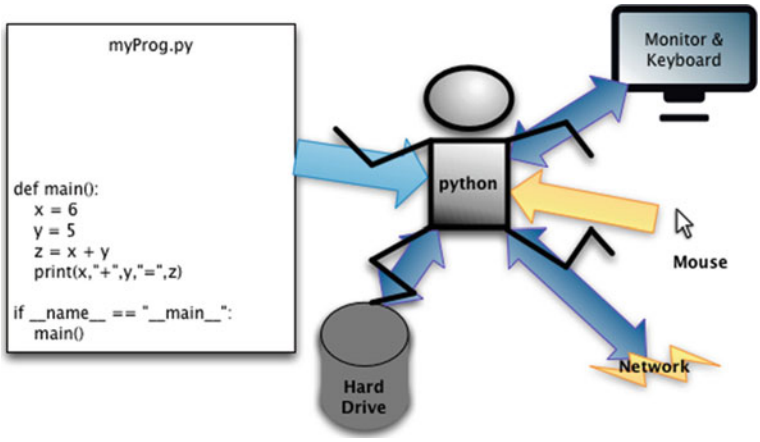

Fig. 3.1 The Python Interpreter

Recursion is the way we will repeat code in this chapter. A recursive function has no need for a *for* or *while* loop.

3.1 Chapter Goals

By the end of this chapter, you should be able to answer these questions.

- How does Python determine the meaning of an identifier in a program?
- What happens to the run-time stack when a function is called?
- What happens to the run-time stack when a function returns from a call?
- What are the two important parts to a recursive function and which part comes first?
- Exactly what happens when a return statement is executed?
- Why should we write recursive functions?
- What are the computational complexities of various recursive functions?

You should also be able to write some simple recursive functions yourself without thinking too hard about how they work. In addition, you should be able to use a debugger to examine the contents of the run-time stack for a recursive function.

3.2 Scope

To form a complete mental picture of how your programs work we should further explore just how the Python interpreter executes a Python program. In the first chapter we explored how references are the things which we name and that references point to objects, which are unnamed. However, we sometimes call an object by the name of the reference that is pointing at it. For instance, if we write:

```
x = 6
```

it means that *x* is a reference that points to an object with a *6* inside it. But sometimes we are careless and just say that *x equals 6*. It is important that you understand that even when we say things like *x equals 6* what we really mean is that *x is a reference that points to an object that contains 6*. You can see why we are careless sometimes. It takes too many words to say what we really mean and as long as everyone understands that references have names and objects are pointed to by references, then we can save the words. The rest of this text will make this assumption at times. When it is really important, we'll make sure we distinguish between references and objects.

Part of our mental picture must include *Scope* in a Python program. Scope refers to a part of a program where a collection of identifiers are visible. Let's look at a simple example program.

3.2.1 Local Scope

Consider the code in Fig. 3.2. In this program there are several scopes. Every colored region of the figure delimits one of those scopes. While executing line 23 of the program in Fig. 3.2 the light green region is called the *Local* scope. The local scope is the scope of the function that the computer is currently executing. When your program is executing a line of code, the scope that surrounds that line of code is called the local scope. When you reference an identifier in a statement in your program, Python first examines the local scope to see if the identifier is defined there, within the local scope. An identifier, *id*, is defined under one of three conditions.

- A statement like $id = \ldots$ appears somewhere within the current scope. In this case *id* would be a reference to an object in the local scope.
- *id* appears as a parameter name of the function in the current scope. In this case *id* would be a reference to an object that was passed to the current function as an argument.
- *id* appears as a name of a function or class through the use of a function *def* or *class* definition within the current scope.

While Python is executing line 23 in Fig. 3.2, the reference *val* is defined within its local scope. If Python finds *id* in the local scope, it looks up the corresponding value and retrieves it. This is what happens when *val* is encountered on line 23. The object that is referenced by *val* is retrieved and returned.

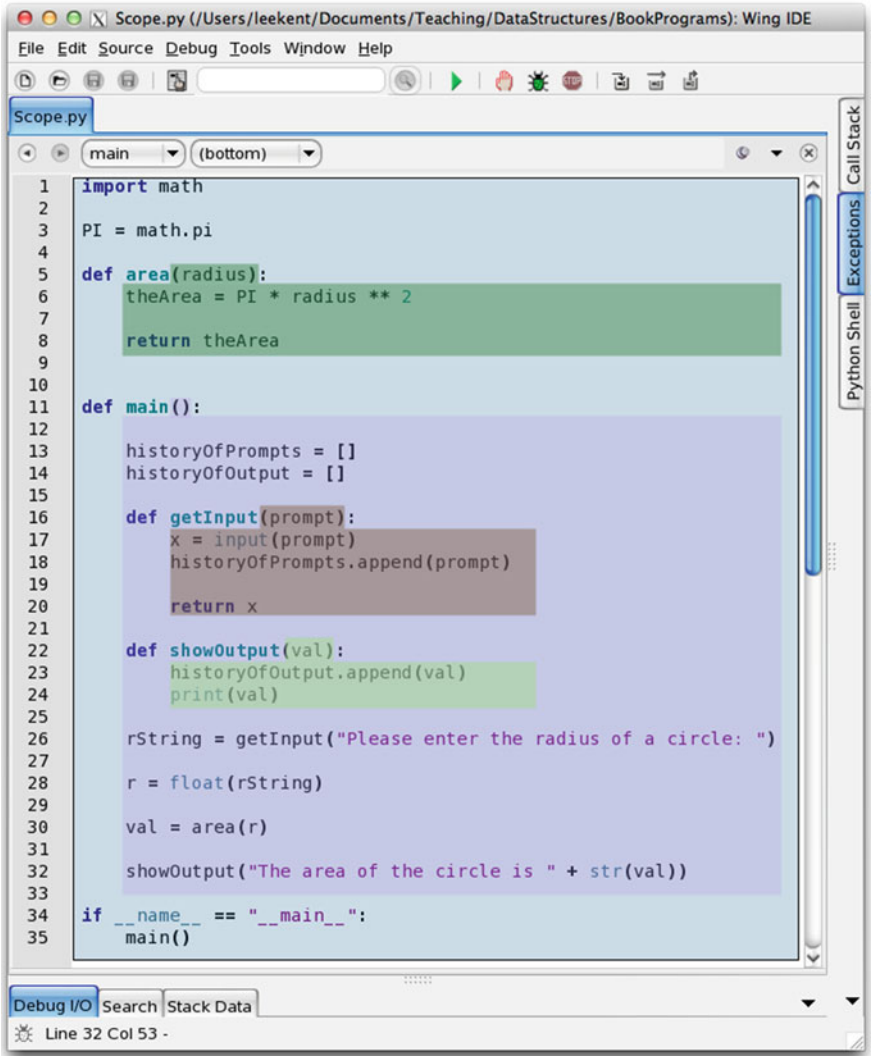

Fig. 3.2 Scopes within a Simple Program

3.2.2 Enclosing Scope

If Python does not find the reference *id* within the local scope, it will examine the *Enclosing* scope to see if it can find *id* there. In the program in Fig. 3.2, while Python is executing the statement on line 23, the enclosing scope is the purple region of the program. The identifiers defined in this enclosing scope include *historyOfPrompts*, *historyOfOutput*, *rString*, *r*, *val*, *getInput*, and *showInput*. Notice that function names are included as identifiers. Again, Python looks for the identifier using the same

conditions as defined in Sect. 3.2.1 for the local scope. The identifier must be defined using *id* = …, it must be a parameter to the enclosing function, or it must be an identifier for a class or function definition in the enclosing scope's function. On line 23, when Python encounters the identifier *historyOfOutput* it finds that identifier defined in the enclosing scope and retrieves it for use in the call to the append method.

Which scope is local depends on where your program is currently executing. When executing line 23, the light green region is the local scope. When executing line 18 the brown region is the local scope. When executing line 14 or line 26 the purple region is the local scope. When executing line 6 the darker green region is the local scope. Finally, when executing line 1 or 3 the blue region is the local scope. The local scope is determined by where your program is currently executing.

Scopes are nested. This means that each scope is nested inside another scope. The final enclosing scope of a module is the module itself. Each module has its own scope. The blue region of Fig. 3.2 corresponds to the module scope. Identifiers that are defined outside of any other functions, but inside the module, are at the module level. The reference *PI* in Fig. 3.2 is defined at the module level. The functions *area* and *main* are also defined at the module level scope.

While executing line 23 of the program in Fig. 3.2 the identifier *val* is defined in the local scope. But, *val* is also defined in the enclosing scope. This is acceptable and often happens in Python programs. Each scope has its own copy of identifiers. The choice of which *val* is visible is made by always selecting the innermost scope that defines the identifier. While executing line 23 of the program in Fig. 3.2 the *val* in the local scope is visible and the *val* in the enclosing scope is hidden. This is why it is important that we choose our variable names and identifiers carefully in our programs. If we use an identifier that is already defined in an outer scope, we will no longer be able to access it from an inner scope where the same identifier is defined.

It is relatively easy to determine all the nested scopes within a module. Every function definition (including the definition of methods) within a module defines a different scope. The scope never includes the function name itself, but includes its parameters and the body of the function. You can follow this pattern to mentally draw boxes around any scope so you know where it begins and ends in your code.

3.2.3 Global Scope

Using Python it is possible to define variables at the *Global* level. Generally this is a bad programming practice and we will not do this in this text. If interested you can read more about global variables in Python online. But, using too many global variables will generally lead to name conflicts and will likely lead to unwanted side effects. Poor use of global variables contributes to spaghetti code which is named for the big mess you would have trying to untangle it to figure out what it does.

3.2.4 Built-In Scope

The final scope in Python is the *Built-In* scope. If an identifier is not found within any of the nested scopes within a module and it is not defined in the global scope, then Python will examine the built-in identifiers to see if it is defined there. For instance, consider the identifier *int*. If you were to write the following:

```
x = int("6")
```

Python would first look in the local scope to see if *int* were defined as a function or variable within that local scope. If *int* is not found within the local scope, Python would look in all the enclosing scopes starting with the next inner-most local scope and working outwards from there. If not found in any of the enclosing scopes, Python would then look in the global scope for the *int* identifier. If not found there, then Python would consult the *Built-In* scope, where it would find the *int* class or type.

With this explanation, it should now be clear why you should not use identifiers that already exist in the built-in scope. If you use *int* as an identifier you will not be able to use the *int* from the built-in scope because Python will find *int* in a local or enclosing scope first.

3.2.5 LEGB

Mark Lutz, in his book *Learning Python* [6], described the rules of scope in Python programs using the LEGB acronym. This acronym, standing for *Local*, *Enclosing*, *Global*, and *Built-In* can help you memorize the rules of scope in Python. The order of the letters in the acronym is important. When the Python interpreter encounters an identifier in a program, it searches the local scope first, followed by all the enclosing scopes from the inside outward, followed by the global scope, and finally the built-in scope.

3.3 The Run-Time Stack and the Heap

As we learned in the last section, the parameters and body of each function define a scope within a Python program. The parameters and variables defined within the local scope of a function must be stored someplace within the RAM of a computer. Python splits the RAM up into two parts called the *Run-time Stack* and the *Heap*.

The run-time stack is like a stack of trays in a cafeteria. Most cafeterias have a device that holds these trays. When the stack of trays gets short enough a spring below the trays pops the trays up so they are at a nice height. As more trays are added to the stack, the spring in this device compresses and the stack pushes down. A *Stack* in Computer Science is similar in many ways to this kind of device. The run-time stack is a stack of *Activation Records*. The Python interpreter *pushes* an activation

record onto the run-time stack when a function is called. When a function returns the Python interpreter *pops* the corresponding activation record off the run-time stack.

Python stores the identifiers defined in the local scope in an activation record. When a function is called, a new scope becomes the local scope. At the same time a new activation record is pushed onto the run-time stack. This new activation record holds all the variables that are defined within the new local scope. When a function returns its corresponding activation record is popped from the run-time stack.

The *Heap* is the area of RAM where all objects are stored. When an object is created it resides in the heap. The run-time stack never contains objects. References to objects are stored within the run-time stack and those references point to objects in the heap.

Consider the program in Fig. 3.2. When the Python interpreter is executing lines 23 and 24 of the program, the run-time stack looks as it does in Fig. 3.3. There are three activation records on the run-time stack. The first activation record pushed onto the run-time stack was for the *module*. When the module first began executing, the Python interpreter went through the module from top to bottom and put any variable definitions in the module scope into the activation record for the module. In this program that consisted of the reference *PI* to the value 3.14159.

Then, at the end of the module the *if* statement called the *main* function. This caused the Python interpreter to push the activation record for the main function. The variables defined within the main function include *historyOfPrompts*, *historyOfOutput*, *rString*, *r*, and *val*. Each of these appear within the activation record for the main function.

As the main function began executing it called the *getInput* function. When that call occurred there was an activation record pushed for the function call. That activation record contained the *prompt* and *x* variables. This activation record does not appear in the figure because by the time we execute line 23 and 24 of the program the Python interpreter has already returned from the *getInput* function. When the interpreter returned from the function call the corresponding activation record was popped from the run-time stack.

Finally, the program calls the *showOutput* function on line 26 and execution of the function begins. An activation record for the *showOutpout* function call was pushed onto the run-time stack when *showOutput* was called. The references local to that scope, which includes just the *val* variable, were stored the activation record for this function call.

You can run this example program using Wing or some other IDE. The code for it appears in Sect. 20.2. When you use the Wing IDE to run this program you can stop the program at any point and examine the run-time stack. For instance, Fig. 3.4 shows Wing in the midst of running this program. A breakpoint has been set on line 24 to stop the program. The tab at the bottom of the Wing IDE window shows the *Stack Data*. This is the run-time stack.

Right below the *Stack Data* tab there is a combination box that currently displays *showOutput*(): *Scope.py, line 24*. This combo box lets you pick from the activation record that is currently being displayed. If you pick a different activation record, its contents will be displayed directly below it in the Wing IDE.

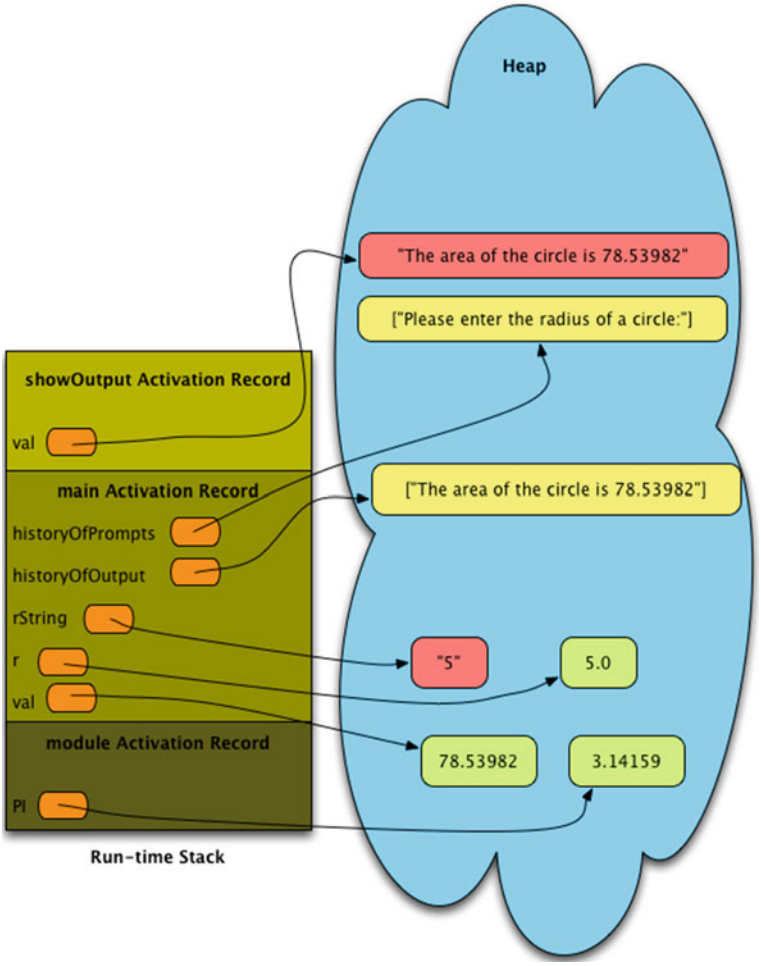

Fig. 3.3 The Run-time Stack and the Heap

One important note should be made here. Figure 3.4 shows *historyOfOutput* as a local variable in the *showOutput* function. This is not really the case, because the *historyOfOutput* reference is not defined within the local scope of the *showOutput* function. However, due to the way Python is implemented the reference for this variable shows up in the activation record for *showOutput* because it is being referenced from this scope. But, the reference to *historyOfOutput* in the activation record for *showOutput* and the reference called *historyOfOutput* in the *main* activation record point at the same object so no real harm is done. The important thing to note is that the Wing IDE is correct in showing the *historyOfOutput* variable as a local variable in this activation record since this is a reflection of Python's implementation and not due to a bug in Wing IDE 101.

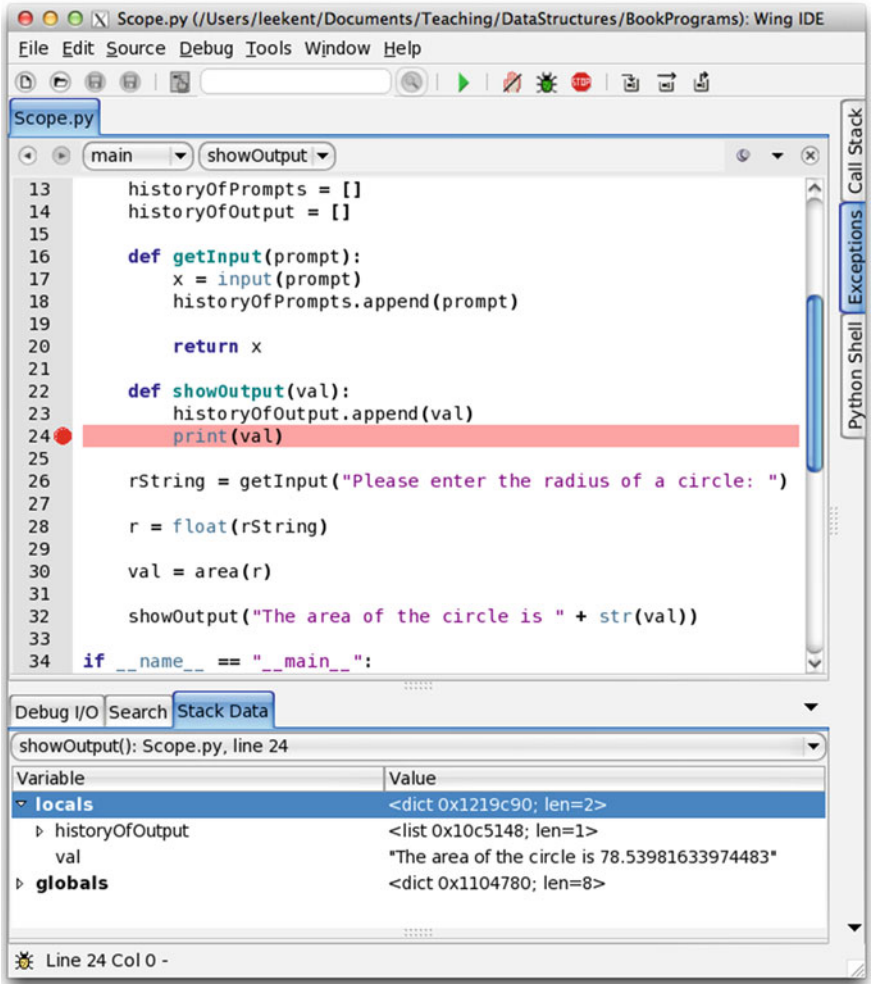

Fig. 3.4 The Wing IDE Showing the Run-time Stack

3.4 Writing a Recursive Function

A recursive function is simply a function that calls itself. It's really very simple to write a recursive function, but of course you want to write recursive functions that actually do something interesting. In addition, if a function just kept calling itself it would never finish. Actually, it would finish when run on a computer because we just learned that every time you call a function, an activation record is pushed on the run-time stack. If a recursive function continues to call itself over and over it will

eventually fill up the run-time stack and you will get a stack overflow error when running such a program.

To prevent a recursive function from running forever, or overflowing the run-time stack, every recursive function must have a base case, just like an inductive proof must have a base case. There are many similarities between inductive proofs and recursive functions. The base case in a recursive function must be written first, before the function is called recursively.

Now, wrapping your head around just how a recursive function works is a little difficult at first. Actually, understanding *how* a recursive function works isn't all that important. When writing recursive functions we want to think more about *what* it does than *how* it works. It doesn't pay to think too hard about *how* recursive functions work, but in fact even that will get much easier with some practice.

When writing a recursive function there are four rules that you adhere to. These rules are not negotiable and will ensure that your recursive function will eventually finish. If you memorize and learn to follow these rules you will be writing recursive functions in no time. The rules are:

1. Decide on the name of your function and the arguments that must be passed to it to complete its work as well as what value the function should return.
2. Write the base case for your recursive function first. The base case is an *if* statement that handles a very simple case in the recursive function by returning a value.
3. Finally, you must call the function recursively with an argument or arguments that are smaller in some way than the parameters that were passed to the function when the last call was made. The argument or arguments that get smaller are the same argument or arguments you examined in your base case.
4. Look at a concrete example. Pick some values to try out with your recursive function. Trust that the recursive call you made in the last step works. Take the result from that recursive call and use it to form the result you want your function to return. Use the concrete example to help you see how to form that result.

We'll do a very simple example to begin with. In the last chapter we proved the following.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

So, if we wanted to compute the sum of the first n integers, we could write a Python program as shown in Sect. 3.4.1.

3.4.1 Sum of Integers

```
1  def sumFirstN(n):
2      return n * (n+1) // 2
3
4  def main():
```

```
    x = int(input("Please enter a non-negative integer: "))

    s = sumFirstN(x)

    print("The sum of the first", x, "integers is", str(s)+".")

if __name__ == "__main__":
    main()
```

In this case, this would be the best function we could write because the complexity of the *sumFirstN* function is O(1). This means the time it takes to execute this function is not dependent on the size of the data, n. However, to illustrate a recursive function, let's go back to the definition of summation. The definition for summation has two parts. First, the base case of the definition.

$$\sum_{i=1}^{0} i = 0$$

The recursive part of the definition is as follows. This is what we call a recursive definition because it is defined in terms of itself. Notice that the recursive definition is defined in terms of a smaller n, in this case $n - 1$. The summation to $n - 1$ is our recursive call and it will work. If we want to compute the sum of the first 5 integers, then the recursive call computes $1 + 2 + 3 + 4$ to give us 10. Adding n will give use 15, the result we want.

$$\sum_{i=1}^{n} i = \left(\sum_{i=1}^{n-1} i\right) + n$$

The two parts of this recursive definition can be translated directly into a recursive function in Python. The recursive definition is given in Sect. 3.4.2.

3.4.2 Recursive Sum of Integers

```
def recSumFirstN(n):
    if n == 0:
        return 0
    else:
        return recSumFirstN(n-1) + n

def main():
    x = int(input("Please enter a non-negative integer: "))

    s = recSumFirstN(x)

    print("The sum of the first", x, "integers is", str(s)+".")

if __name__ == "__main__":
    main()
```

The *recSumFirstN* function in the code of Sect. 3.4.2 is recursive. It calls itself with a smaller value and it has a base case that comes first, so it is well-formed. There is

one thing that we might point out in this recursive function. The *else* is not necessary. When the Python interpreter encounters a **return** statement, the interpreter returns immediately and does not execute the rest of the function. So, in Sect. 3.4.2, if the function returns 0 in the *then* part of the *if* statement, the rest of the function is not executed. If *n* is not zero, then we want to execute the code on the *else* statement. This means we could rewrite this function as shown in Sect. 3.4.3.

3.4.3 No Else Needed

```
1  def recSumFirstN(n):
2      if n == 0:
3          return 0
4
5      return recSumFirstN(n-1) + n
```

The format of the code in Sect. 3.4.3 is a common way to write recursive functions. Sometimes a recursive function has more than one base case. Each base case can be handled by an *if* statement with a return in it. The recursive case does not need to be in an else when all base cases result in a return. The recursive case comes last in the recursive function definition.

3.5 Tracing the Execution of a Recursive Function

Early in this chapter you were given the mandate "Don't think too hard" when writing a recursive function. Understanding exactly *how* a recursive function works may be a bit difficult when you are first learning about them. It may help to follow the execution of a recursive function in an example. Consider the program in the previous section. Let's assume that the user entered the integer 4 at the keyboard. When this program begins running it will have an activation record on the run-time stack for the *module* and the *main* function.

When the program gets to line 10 in the code of Sect. 3.4.2, where the *recSumFirstN* function is first called, a new activation record will be pushed for the function call, resulting in three activation records on the run-time stack. The Python interpreter then jumps to line 2 with *n* pointing at the number 4 as shown in the picture of Fig. 3.5. Execution of the function proceeds. The value of *n* is not zero, so Python executes line 5 where there is another function call to *recSumFirstN*. This causes the Python interpreter to push another activation record on the run-time stack and the interpreter jumps to line 2 again. This time the value of *n* is 3. But again, this is not zero, so line 5 is executed and another activation record is pushed with a new value of 2 for *n*. This repeats two more times for values of 1 and 0 for *n*.

The important thing to note in this program execution is that there is one copy of the variable *n* for each recursive function call. An activation record holds the local variables and parameters of all variables that are in the local scope of the function.

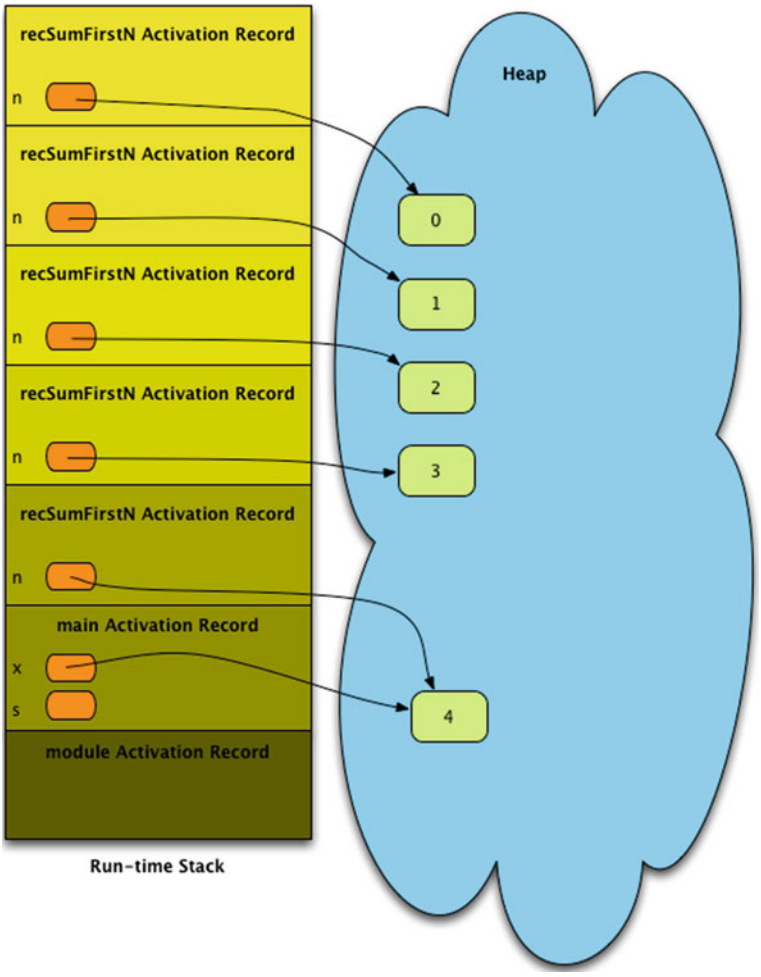

Fig. 3.5 The Run-time Stack of a Recursive Function Call

Each time the function is called a new activation record is pushed and a new copy of the local variables is stored within the activation record. The picture in Fig. 3.5 depicts the run-time stack at its deepest point.

When execution of the function gets to the point when n equals 0, the Python interpreter finds that n equals 0 on line 2 of the code. It is at this point that the *sumFirstN* function returns its first value. It returns 0 to the previous function call where n was 1. The return occurs on line 5 of the code. The activation record for the function call when n was 0 is popped from the run-time stack. This is depicted in Fig. 3.6 by the shading of the activation record in the figure. When the function returns the space for the activation record is reclaimed for use later. The shaded

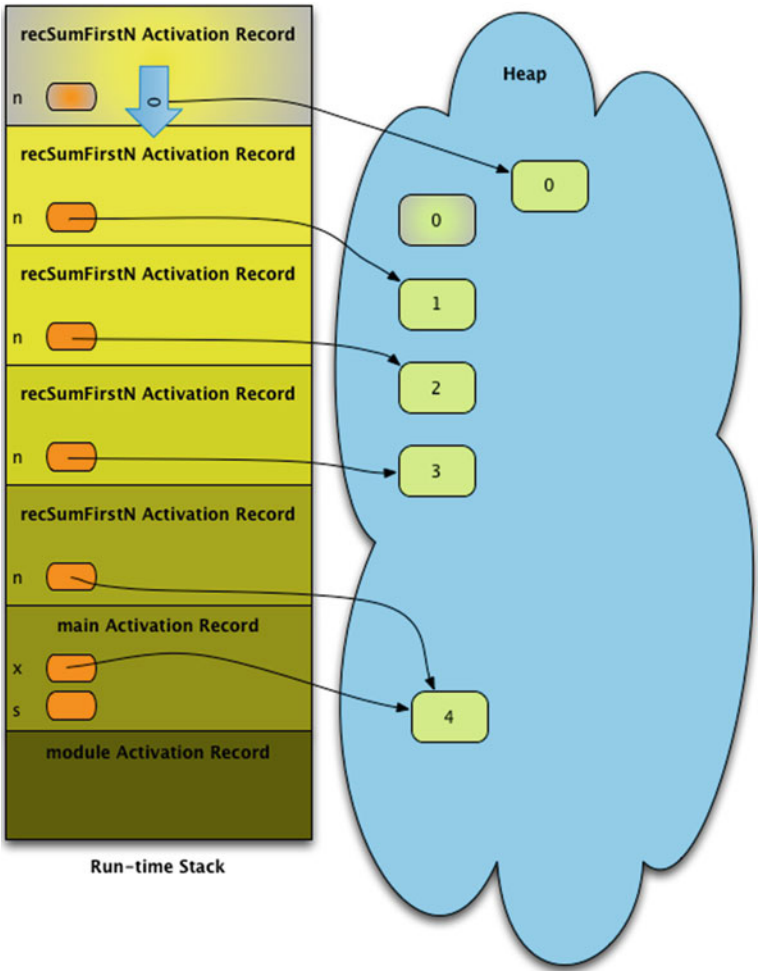

Fig. 3.6 The First Return from recSumFirstN

object containing 0 on the heap is also reclaimed by the garbage collector because there are no references pointing at it anymore.

After the first return of the *RecSumFirstN*, the Python interpreter returns to line 5 in the previous function call. But, this statement contains a return statement as well. So, the function returns again. Again, it returns to line 5, but this time with a value of 1. The function returns again, but with a value of 3 this time. Again, since it returned to line 5, the function returns again with a value of 6. Finally, once again the function returns, this time with a value of 10. But this time the *recSumFirstN* function returns to line 10 of the main function where *s* is made to point to the value of 10. This is depicted in Fig. 3.7.

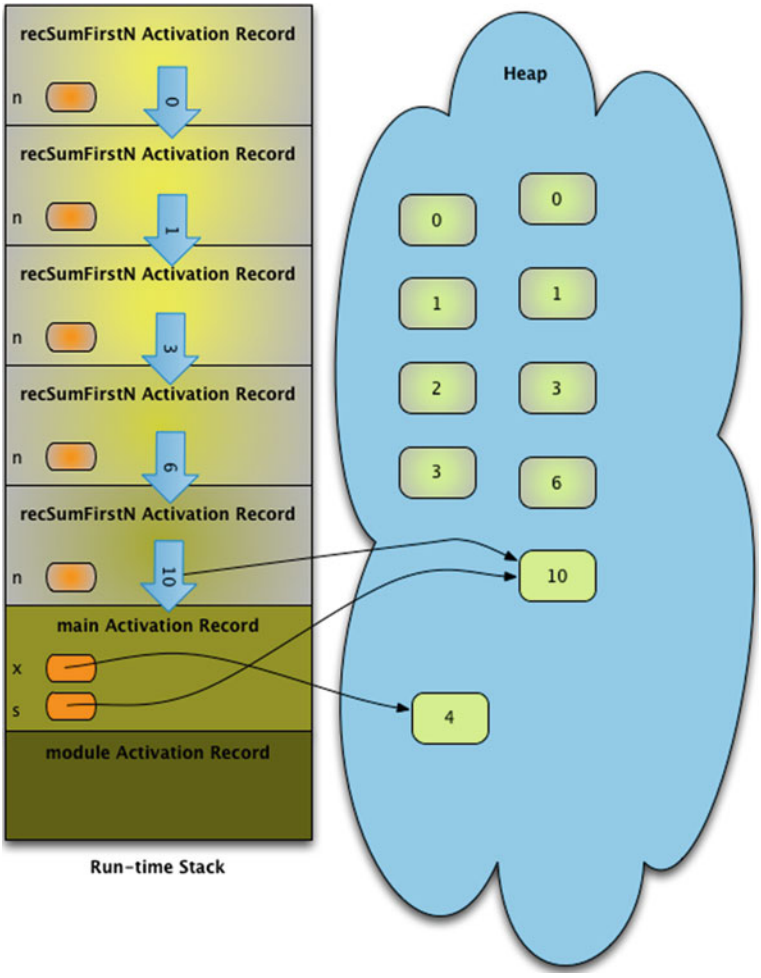

Fig. 3.7 The Last Return from recSumFirstN

The program terminates after printing the 10 to the screen and returning from the *main* function after line 12 and from the *module* after line 15. The importance of this example is to illustrate that each recursive call to *recSumFirstN* has its own copy of the variable *n* because it is local to the scope of the *recSumFirstN* function. Each time the function is called, the local variables and parameters are copied into the corresponding activation record. When a function call returns, the corresponding activation record is popped off the run-time stack. This is how a recursive function is executed.

3.6 Recursion in Computer Graphics

Recursion can be applied to lots of different problems including sorting, searching, drawing pictures, etc. The program given in Sect. 3.6.1 draws a spiral on the screen as shown in Fig. 3.8.

3.6.1 Recursive Spiral

```
import turtle

def drawSpiral(t, length, color, colorBase):
    #color is a 24 bit value that is changing a bit
    #each time for a nice color effect
    if length == 0:
        return

    # add 2^10 to the old color modulo 2^24
    # the modulo 2^24 prevents the color from
    # getting too big.
    newcolor = (int(color[1:],16) + 2**10)%(2**24)

    # find the color base integer value
    base = int(colorBase[1:],16)

    # now if the new color is less than the base
    # add the base modulo 2^24.
    if newcolor < base:
        newcolor = (newcolor + base)%(2**24)

    # let newcolor be the hex string after conversion.
    newcolor = hex(newcolor)[2:]

    # add a pound sign and zeroes to the front so it
    # is 6 characters long plus the pound sign for a
    # proper color string.
    newcolor = "#"+("0"*(6-len(newcolor)))+newcolor

    t.color(newcolor)
    t.forward(length)
    t.left(90)

    drawSpiral(t, length-1, newcolor, colorBase)

def main():
    t = turtle.Turtle()
    screen = t.getscreen()
    t.speed(100)
    t.penup()
    t.goto(-100,-100)
    t.pendown()

    drawSpiral(t, 200, "#000000", "#ff00ff")

    screen.exitonclick()

```

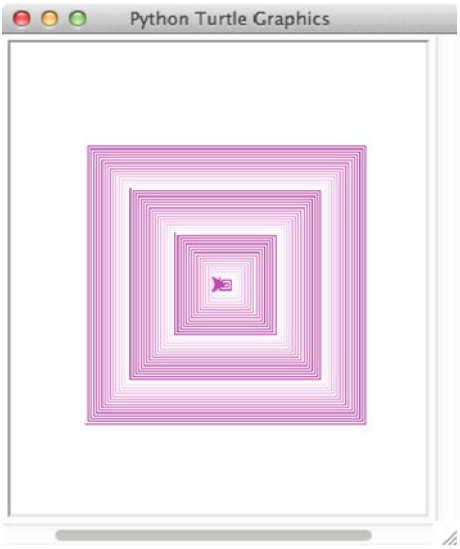

Fig. 3.8 A Spiral Image

```
48 if __name__ == "__main__":
49     main()
```

In this program the *drawSpiral* function is recursive. It has a base case that is written first: when the length of the side is zero it exits. It calls itself on something smaller: the new length passed to it is the old length minus one. The newcolor formula is perhaps the most complex part of the code. There is some slicing going on there to convert the color string from a hexadecimal string to an integer so 1024 can be added, modulo 2 to the 24th. Then it must be converted back to a hexadecimal color string with the "#ffffff" format. The program draws a spiral like the one pictured in Fig. 3.8.

Notice that this recursive function does not return anything. Most recursive functions do return a value. This one does not because the purpose of the function is to draw a spiral. It has a side-effect instead of returning a value.

3.7 Recursion on Lists and Strings

Recursive functions can be written for many different purposes. Many problems can be solved by solving a simpler problem and then applying that simpler solution recursively. For instance, consider trying to write a function that returns the reverse of a list. If we wrote this non-recursively, we might write it as follows.

3.7.1 List Recursion

```
1 def revList(lst):
2     accumulator = []
```

```
3
4       for x in lst:
5           accumulator = [x] + accumulator
6
7       return accumulator
8
9   def main():
10      print(revList([1,2,3,4]))
11
12  if __name__ == "__main__":
13      main()
```

When run, this program prints [4, 3, 2, 1] to the screen. The code in Sect. 3.7.1 uses the *accumulator pattern* to solve the problem of reversing a list. This is a pattern you have probably used before if you first learned to program imperatively. If we think about the problem recursively, we would first consider how to reverse a very simple list, say the empty list. The reverse of the empty list is just the empty list.

Once we have solved the problem for a very simple list, we can assume that if we call a recursive reverse function on something smaller (i.e. a shorter list), it will work. So, then to complete a recursive solution, we have only to piece our solution together. A recursive solution to reversing a list is found in Sect. 3.7.2.

3.7.2 Reversing a List

```
1   def revList(lst):
2       # Here is the base case
3       if lst == []:
4           return []
5
6       # The rest of this function is the recursive case.
7       # This works because we called it on something smaller.
8       # The lst[1:] is a slice of all but the first item in lst.
9       restrev = revList(lst[1:])
10      first = lst[0:1]
11
12      # Now put the pieces together.
13      result = restrev + first
14
15      return result
16
17
18  def main():
19      print(revList([1,2,3,4]))
20
21  if __name__ == "__main__":
22      main()
```

You can write recursive functions that work with strings too. Strings and lists are both sequences. In the code of Sect. 3.7.2 we made sure we recursively called our function on something smaller. The same is true when working with strings. A string reverse function is given in Sect. 3.7.3.

3.7.3 Reversing a String

```
def revString(s):
    if s == "":
        return ""

    restrev = revString(s[1:])
    first = s[0:1]
    # Now put the pieces together.
    result = restrev + first

    return result


def main():
    print(revString("hello"))

if __name__ == "__main__":
    main()
```

Notice the similarity of these two functions. The functions are nearly identical. That's because the recursive definition of reverse did not change. The only change is that we must use the string concatenation operator instead of the list concatenation operator and the empty string instead of the empty list.

3.7.4 Another Version of Reverse

```
def revList2(lst):

    def revListHelper(index):
        if index == -1:
            return []

        restrev = revListHelper(index-1)
        first = [lst[index]]

        # Now put the pieces together.
        result = first + restrev

        return result

    # this is the one line of code for the
    # revList2 function.
    return revListHelper(len(lst)-1)


def main():
    print(revList2([1,2,3,4]))

if __name__ == "__main__":
    main()
```

The examples in Sects. 3.7.2 and 3.7.3 used slicing to make the list or string smaller on each recursive call. It is possible to make a list or string *smaller* without actually making it physically smaller. Using an index to keep track of your position

within a list can serve to make the list or string smaller. In that case it may be helpful to write a function that calls a *helper* function to do the recursion. Consider the program in Sect. 3.7.4. This code uses a nested helper function called *revListHelper* to do the actual recursion. The list itself does not get smaller in the helper function. Instead, the *index* argument gets smaller, counting down to −1 when the empty list is returned. The *revList2* function contains only one line of code to call the *revListHelper* function.

Because the *revListHelper* function is nested inside *revList2* the helper function is not visible to anything but the *revList2* function since we don't want other programmers to call the helper function except by calling the *revList2* function first.

It is important to note that you don't have to physically make a list or string smaller to use it in a recursive function. As long as indexing is available to you, a recursive function can make use of an index into a list or string and the index can get smaller on each recursive call.

One other thing to note. In this example the index gets smaller by approaching zero on each recursive call. There are other ways for the argument to the recursive function to get *smaller*. For instance, this example could be rewritten so the index grows toward the length of the list. In that case the *distance* between the index and the length of the list is the value that would get smaller on each recursive call.

3.8 Using Type Reflection

Many of the similarities in the two functions of Sects. 3.7.3 and 3.7.2 are due to operator overloading in Python. Python has another very nice feature called *reflection*. Reflection refers to the ability for code to be able to examine attributes about objects that might be passed as parameters to a function. One interesting aspect of reflection is the ability to see what the *type* of an object is. If we write *type*(*obj*) then Python will return an object which represents the *type* of *obj*. For instance, if *obj* is a reference to a string, then Python will return the *str type* object. Further, if we write *str*() we get a string which is the empty string. In other words, writing *str*() is the same thing as writing "". Likewise, writing *list*() is the same thing as writing []. Using reflection, we can write one recursive reverse function that will work for strings, lists, and any other sequence that supports slicing and concatenation. A recursive version of reverse that will reverse both strings and lists is provided in Sect. 3.8.1.

3.8.1 Reflection Reverse

```
1 def reverse(seq):
2     SeqType = type(seq)
3     emptySeq = SeqType()
4
```

```
5       if seq == emptySeq:
6           return emptySeq
7
8       restrev = reverse(seq[1:])
9       first = seq[0:1]
10
11      # Now put the pieces together.
12      result = restrev + first
13
14      return result
15
16
17  def main():
18      print(reverse([1,2,3,4]))
19      print(reverse("hello"))
20  if __name__ == "__main__":
21      main()
```

After writing the code in Sect. 3.8.1 we have a polymorphic reverse function that will work to reverse any sequence. It is polymorphic due to reflection and operator overloading. Pretty neat stuff!

3.9 Chapter Summary

In this chapter, you were introduced to some concepts that are important to your understanding of algorithms to be presented later in this text. Understanding how the run-time stack and the heap work to make it possible to call functions in our programs will make you a better programmer. Forming a mental model of how our code works makes it possible to predict what our code will do. Writing recursive functions is also a skill that is important to computer programmers. Here is what you should have learned in this chapter. You should:

- be able to identify the various scopes within a program.
- be able to identify which scope a variable reference belongs to: the local, enclosing, global, or built-in scope. Remember the LEGB rule.
- be able to trace the execution of a program by drawing a picture of the run-time stack and the heap for a program as it executes.
- be able to write a simple recursive function by writing a base case and a recursive case where the function is called with a smaller value.
- be able to trace the execution of a recursive function, showing the run-time stack and heap as it executes.
- understand a little about reflection as it relates to examining *types* in Python code.

3.10 Review Questions

Answer these short answer, multiple choice, and true/false questions to test your mastery of the chapter.

1. What is an interpreter?
2. What is the Python interpreter called?
3. When the Python interpreter sees an identifier, in which scope does it look for the identifier first?
4. What order are the various scopes inspected to see where or if a variable is defined?
5. Pick a sample program from among the programs you have written, preferably a short one, and identify three scopes within it by drawing a box around the scopes.
6. When is an activation record pushed onto the run-time stack?
7. When is an activation record popped from the run-time stack?
8. What goes in the Heap in a computer?
9. What goes in an activation record on the run-time stack?
10. When writing a recursive function, what are the two cases for which you must write code?
11. If a recursive function did not have a base case, what would happen when it was called?
12. What must be true of the recursive call in a recursive function? In other words, what must you ensure when making this recursive call?
13. What does the *type* function return in Python? If you call the *type* function in a program, what aspect of Python are you using?

3.11 Programming Problems

1. Write a recursive function called intpow that given a number, x, and an integer, n, will compute x ^ n. You must write this function recursively to get full credit. Be sure to put it in a program with several test cases to test that your function works correctly.
2. Write a recursive function to compute the factorial of an integer. The factorial of 0 is 1. The factorial of any integer, n, greater than zero is n times the factorial of n−1. Write a program that tests your factorial function by asking the user to enter an integer and printing the factorial of that integer. Be sure your program has a main function. Comment your code with the base case and recursive case in your recursive function.
3. Write a recursive function that computes the length of a string. You cannot use the *len* function while computing the length of the string. You must rely on the function you are writing. Put this function in a program that prompts the user to enter a string and then prints the length of that string.

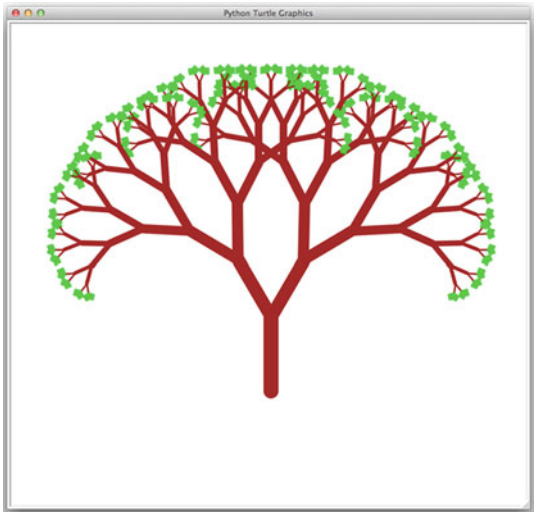

Fig. 3.9 A Tree

4. Write a recursive function that takes a string like "abcdefgh" and returns "badcfehg". Call this function *swap* since it swaps every two elements of the original string. Put this function in a program and call it with at least a few test cases.
5. Write a recursive function that draws a tree. Call your function *drawBranch*. Pass it a turtle to draw with, an angle, and the number of littler branches to draw like the tree that appears in Fig. 3.9. Each time you recursively call this function you can decrease the number of branches and the angle. Each littler branch is drawn at some angle from the current branch so your function can change the angle of the turtle by turning left or right. When your number of branches gets to zero, you can draw a leaf as a little green square. If you make the width of the turtle line thicker for bigger branches and smaller for littler branches, you'll get a nice tree. You might write one more function called drawTree that will set up everything (except the turtle) to draw a nice tree. Put this function in a program that draws at least one tree. **HINT**: In your drawBranch function, after you have drawn the branch (and all sub-branches) you will want to return the turtle to the original position and direction you started at. This is necessary so after calling drawBranch you will know where the turtle is located. If you don't return it to its original position, the turtle will end up stranded out at a leaf somewhere.
6. Write a recursive function that draws a circular spiral. To do this, you'll need to use polar coordinates. Polar coordinates are a way of specifying any point in the plane with an angle and a radius. Zero degrees goes to the right and the angles go counter-clockwise in a circle. With an angle and a radius, any point in the plane can be described. To convert an angle, a, and radius, r, from polar coordinates to Cartesian coordinates you would use sine and cosine. You must import the math module. Then $x = r * math.cos\ (a)$ and $y = r * math.sin\ (a)$.

The *drawSpiral* function will be given a radius for the sprial. To get a circular spiral, every recursive call to the drawSpiral function must decrease the radius just a bit and increase the angle. You convert the angle and the radius to its (x,y) coordinate equivalent and then draw a line to that location. You must also pass an (x,y) coordinate to the *drawSpiral* function for the center point of your spiral. Then, any coordinates you compute will be added to the center (x,y). You can follow the square spiral example in the text. Put this code in a program that draws a spiral to the screen.

7. Write a program to gather performance data for the *reverse* function found in this chapter. Write an XML file in the plot format found in this text to visualize that performance data. Because this function is recursive, keep your data size small and just gather data for string sizes of 1–10. This will help you visualize your result. What is the complexity of this reverse function? Put a comment at the top of your program stating the complexity of reverse in big-Oh notation. Justify your answer by analyzing the code found in the reverse function.
8. Rewrite the program in Sect. 3.7.4 to use an index that approaches the length of the list instead of an index that approaches zero. Then write a main function that thoroughly tests your new reverse function on lists. You must test it on both simple and more complex examples of lists to test it thoroughly.