Defining Functions 5

Functions are something most of us are familiar with from Mathematics. A function g might be defined as

$$g(x) = x^4/4 - x^3/3 - 3x^2$$

When a function is defined this way we can then call the function g with the value 6—usually written $g(6)$—to discover that the value returned by the function would be 144. Of course, we aren't only limited to passing 6 to g. We could pass 0 to g and $g(0)$ would return 0. We could pass any of number into g and compute its result.

The identifier g represents the *definition* of a function and calling a function by writing $g(6)$ is called *function application* or a *function call*. These two concepts are part of most programming languages including Python. In Python, functions can be both *defined* and *called*.

Example 5.1 The function $g(x) = x^4/4 - x^3/3 - 3x^2$ can be defined in Python as shown below. It can also be called as shown here. This program calls g and prints 144.0 to the screen.

```
def g(x):
    return x**4/4.0 - x**3/3.0 - 3 * x * x

print(g(6))
```

To call a function in Python we write $g(6)$ for instance, just they way we do in Mathematics. It means the same thing, too. Executing $g(6)$ means calling the function g with the value 6 to compute the value of the function call. A function in Python can do more than a function in Mathematics. Functions can execute statements as well as return a value. In Mathematics a value is computed and returned. There are no side-effects of calling the function. In Python (and in just about any programming language), there can be side-effects. A function can contain more than one statement just like our programs contain statements.

K.D. Lee, *Python Programming Fundamentals,* Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-84996-537-8_5,

5

Example 5.2 Here is a function that computes and prints a value along with some code that calls the function. Running this program prints "You called computeAndPrint(6,5)" followed by the value 149.0 to the screen. This function is passed two arguments instead of just one.

```
def computeAndPrint(x, y):
    val = x**4/4.0 - x**3/3.0 - 3 * x * x + y
    print("You called computeAndPrint("+str(x)+","+str(y)+")")

    return val

print(computeAndPrint(6,5))
```

5.1 Why Write Functions?

The ability to define our own functions helps programmers in two ways. When you are writing a program if you find yourself writing the same code more than once, it is probably best to define a function with the repeated code in it. Then you can call the function as many times as needed instead of rewriting the code again and again.

It is important that we avoid writing the same code more than once in our programs. Writing code is error-prone. Programmers often make mistakes. If we write the same code more than once and make a mistake in it, we must fix that mistake every place we copied the code. When writing code that will be used commercially, mistakes might not be found until years later. When fixing code that hasn't been looked at for a while it is extremely easy to fix the code in one place and to forget to fix it everywhere.

If we make a mistake in coding a *function*, and then fix the code in the function, we have automatically fixed the code in every spot that uses the function. This principle of modular programming is a very important concept that has been around since the early days of computer programming. Writing code once leads to well-tested functions that work as expected. When we use a well-tested function we can be fairly confident it will work the first time. It also leads to smaller code size, although that is not as much of an issue these days.

Writing functions also helps make our code easier to read. When we use good names for variables and functions in our programs we can read the code and understand what we have written not only as we write it, but years later when we need to look at the code we wrote again. Typically programmers work with a group of three to eight other people. It is important for others in the group to be able to read and understand the code we have written. Writing functions can lead to nice modularized code that is much easier to maintain by you and by others in a group.

5.2 Passing Arguments and Returning a Value

When we write a function we must decide four things:

1. What should our function be called? We should give it a name that makes sense and describes what the function does. Since a function does something, the name of a function is usually a verb or some description of what the function returns. It might be one word or several words long.
2. What should we give to our function? In other words, what arguments will we pass to the function? When thinking about arguments to pass to a function we should think about how the function will be used and what arguments would make it the most useful.
3. What should the function do? What is its purpose? The function needs to have a clearly defined purpose. Either it should return a value or it should have some well-defined side-effect.
4. Finally, what should our function return? The type and the value to be returned should be considered. If the function is going to return a value, we should decide what type of value it should return.

By considering these questions and answering them, we can make sure that our functions make sense before writing them. It does us no good to define functions that don't have a well-defined purpose in our program.

Example 5.3 Consider a program where we are asked to reverse a string. What should the function be called? Probably *reverse*. What should we give to the function? A *string* would make sense. What does *reverse* compute? The reverse of the given string. What should it return? The reversed string. Now we are ready to write the function.

```
def reverse(s):

    # Use the Accumulator Pattern
    result = ""
    for c in s:
        result = c + result

    return result

t = input("Please enter a string: ")
print("The reverse of", t, "is", reverse(t))
```

It is important to decide the type of value *returned* from a function and the types of the arguments *given* to a function. The words *returned* and *given* are words that give us a clue about what the function should look like and what it might do. When presented with a specification for a function look for these words to help you identify what you need to write.

The word *parameter* refers to the identifier used to represent the value that is passed as an *argument* to the function. Sometimes the parameter is called a formal parameter. When a function is called, it is passed an argument as in $g(6)$ where 6 is the argument. When the function is applied the parameter called x takes on the value of 6. If it is called as $g(5)$ then the parameter x takes on the value 5. In this way we can write the function once and it will work for any argument passed to the function. In Example 5.3 the argument is the value that `t` refers to, the value entered by the user when the program is run. The parameter, `s`, takes on the value that t refers to when the function is called in the print statement. The parameter passing mechanism makes it possible for us to write a function once and use it in many different places in our program with many different values passed in.

Practice 5.1 Write a function called explode that given a string returns a list of the characters of the string.

Practice 5.2 Write a function called implode that given a list of characters, or strings, returns a string which is the concatenation of those characters, or strings.

5.3 Scope of Variables

When writing functions it is important to understand scope. Scope refers to the area in a program where a variable is defined. Normally, a variable is defined after it has been assigned a value. You cannot reference a variable until it has been assigned a value.

Practice 5.3 The following program has an run-time error in it. Where does the error occur? Be very specific.

```
x = x + 1
x = 6
print(x)
```

When we define functions there are several identifiers we write. First, the name of the function is written. Like variables, a function identifier can be used after it is defined. In

Example 5.3 you will notice that the function is defined at the top of the program and the function is called on the last line of the program. A function must be defined before it is used.

However, the variables *s*, *c*, and *result* are not available where *reverse*(*t*) is called. This is what we want to happen and is due to something called *scope*. The *scope* of a variable refers to the area in a program where it is defined. There are several scopes available in a Python program. Mark Lutz describes the rules of scope in Python with what he calls the *LEGB* rule [3]. Memorizing the acronym *LEGB* will help you memorize the scope rules of Python.

The *LEGB* rule refers to *Local Scope*, *Enclosing Scope*, *Global Scope*, and *Built-in Scope*. Local scope extends for the body of a function and refers to anything indented in the function definition. Variables, including the parameter, that are defined in the body of a function are local to that function and cannot be accessed outside the function. They are *local variables*.

The *enclosing scope* refers to variables that are defined outside a function definition. If a function is defined within the scope of other variables, then those variables are available inside the function definition. The variables in the enclosing scope are available to statements within a function.

Example 5.4 While this is not good coding practice, the following code illustrates the enclosing scope. The *values* variable keeps track of all the arguments passed to the reverse function.

```
def reverse(s):

    values.append(s)

    # Use the Accumulator Pattern
    result = ""
    for c in s:
        result = c + result

    return result

# The values variable is defined in the enclosing scope of
# the reverse function.
values = []

t = input("Please enter a string: ")
while t.strip() != "":
    print("The reverse of", t, "is", reverse(t))
    t = input("Enter another string or press enter to quit: ")

print("You reversed these strings:")
for val in values:
    print(val)
```

Accessing a variable in the enclosing scope can be useful in some circumstances, but is not usually done unless the variable is a constant that does not change in a program. In the program above the *values* variable is accessed by the reverse function on the first line of its body. This is an example of using *enclosing* scope. However, the next example, while it does almost the same thing, has a problem.

Example 5.5 Here is an example of almost the same program. Instead of using the mutator method *append* the list concatenation (i.e. the +) operator is used to append the value *s* to the list of values.

```
def reverse(s):

    # The following line of code will not work.
    values = values + [s]

    # Use the Accumulator Pattern
    result = ""
    for c in s:
        result = c + result

    return result

# The values variable is defined in the enclosing scope of
# the reverse function.
values = []

t = input("Please enter a string: ")
while t.strip() != "":
    print("The reverse of", t, "is", reverse(t))
    t = input("Enter another string or press enter to quit: ")

print("You reversed these strings:")
for val in values:
    print(val)
```

The code in Example 5.5 does not work because of a subtle issue in Python. A new variable, say v, is defined in Python anytime $v = \cdots$ is written. In Example 5.5 the first line of the reverse function is $values = values + [s]$. As soon as $values = \cdots$ is written, there is a new local variable called *values* that is defined in the scope of the *reverse* function. That means there are two variables called *values*: one defined in reverse and one defined outside of reverse. The problem occurs when the right-hand side of $values = values + [s]$ is evaluated. Which *values* is being concatenated to $[s]$. Is it the local or the enclosing *values*. Clearly, we would like it to be the enclosing *values* variable. But, it is not. Local scope overrides enclosing scope and the program in Example 5.5 will complain that *values* does not yet have a value on the first line of the reverse function's body.

The problem with Example 5.5 can be fixed by declaring the *values* variable to be *global*. When applied to a variable, global scope means that there should not be a local

copy of a variable made, even when it appears on the left hand side of an assignment statement.

Example 5.6 The string concatenation operator can still be used if the *values* variable is declared to be global in the *reverse* function.

```
def reverse(s):
    global values

    #values.append(s)
    values = values + [s]

    # Use the Accumulator Pattern
    result = ""
    for c in s:
        result = c + result

    return result

# The values variable is defined in the enclosing scope of
# the reverse function.
values = []

t = input("Please enter a string: ")
while t.strip() != "":
    print("The reverse of", t, "is", reverse(t))
    t = input("Enter another string or press enter to quit: ")

print("You reversed these strings:")
for val in values:
    print(val)
```

Example 5.6 demonstrates the use of the global scope. The use of the *global* keyword forces Python to use the variable in the enclosing scope even when it appears on the left hand side of an assignment statement.

The final scope rule is the built-in scope. The built-in scope refers to those identifiers that are built-in to Python. For instance, the *len* function can be used anywhere in a program to find the length of a sequence.

The one gotcha with scope is that local scope trumps the enclosing scope, which trumps the global scope, which trumps the built-in scope. Hence the LEGB rule. First local scope is scanned for the existence of an identifier. If that identifier is not defined in the local scope, then the enclosing scope is consulted. Again, if the identifier is not found in enclosing scope then the global scope is consulted and finally the built-in scope. This does have some implications in our programs.

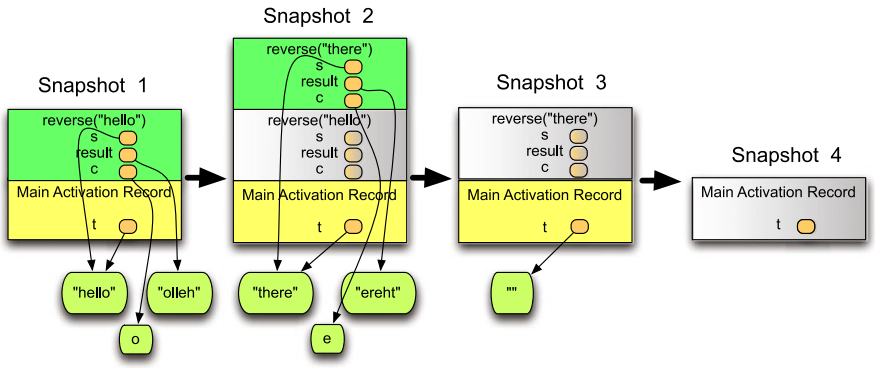

Fig. 5.1 The run-time stack

Practice 5.4 The following code does not work. What is the error message? Do you see why? Can you suggest a way to fix it?

```
def length(L):
    len = 1
    for i in range(len(L)):
        len = len + 1

    return len

print(length([1,2,3]))
```

5.4 The Run-time Stack

The run-time stack is a data structure that is used by Python to execute programs. Python needs this run-time stack to maintain information about the state of your program as it executes. A *stack* is a data structure that lets you push and pop elements. You push elements onto the top of the stack and you pop elements from the top of the stack. Think of a stack of trays. You take trays off the top of the stack in a cafeteria. You put clean trays back on the top of the stack. A stack is a *first in/first out* data structure. Stacks can be created to hold a variety of different types of elements. The *run-time stack* is a stack of *activation records*.

An activation record is an area of memory that holds a copy of each variable that is defined in the local scope of a function while it is executing. As we learned in Example 5.5,

a variable is defined when it appears on the left-hand side of an equals sign. Formal parameters of a function are also in the local scope of the function.

Example 5.7 In this example code the reverse function is called repeatedly until the user enters an empty string (just presses enter) to end the program. Each call to reverse pushes an activation record on the stack.

```
def reverse(s):

    # Use the Accumulator Pattern
    result = ""
    for c in s:
        result = c + result

    return result

t = input("Please enter a string: ")
while t.strip() != "":
     print("The reverse of", t, "is", reverse(t))
     t = input("Enter another string or press enter to quit: ")
```

In Example 5.7, each time the reverse function returns to the main code the activation record is popped. Assuming the user enters the string "hello", snapshot 1 of Fig. 5.1 shows what the run-time stack would look like right before the result is returned from the function call.

In snapshot 2, the activation record for reverse ("hello") had been popped from the stack but is shown grayed out in snapshot 2 to make it clear that the top activation record is a *new* activation record. Snapshot 2 was taken right before the *return result* statement was executed for the second call to reverse.

Snapshot 3 shows what the run-time stack looks like after returning from the second call to reverse. Again, the grayed out activation record is not there, but is shown to emphasize that it is popped when the function returns. Finally, snapshot 4 shows what happens when the main code exits, causing the last activation record to be popped.

Each activation record holds a copy of the local variables and parameters that were passed to the function. Local variables are those variables that appear on the left-hand side of an equals sign in the body of the function or appear as parameters to the function. Recall that variables are actually references in Python, so the references or variables point to the actual values which are not stored in the activation records.

The run-time stack is absolutely critical to the implementation of modern programming languages. Its existence makes it possible for a function to execute and return independently of where it is called. This independence between functions and the code that calls them is crucial to making functions useful in our programs.

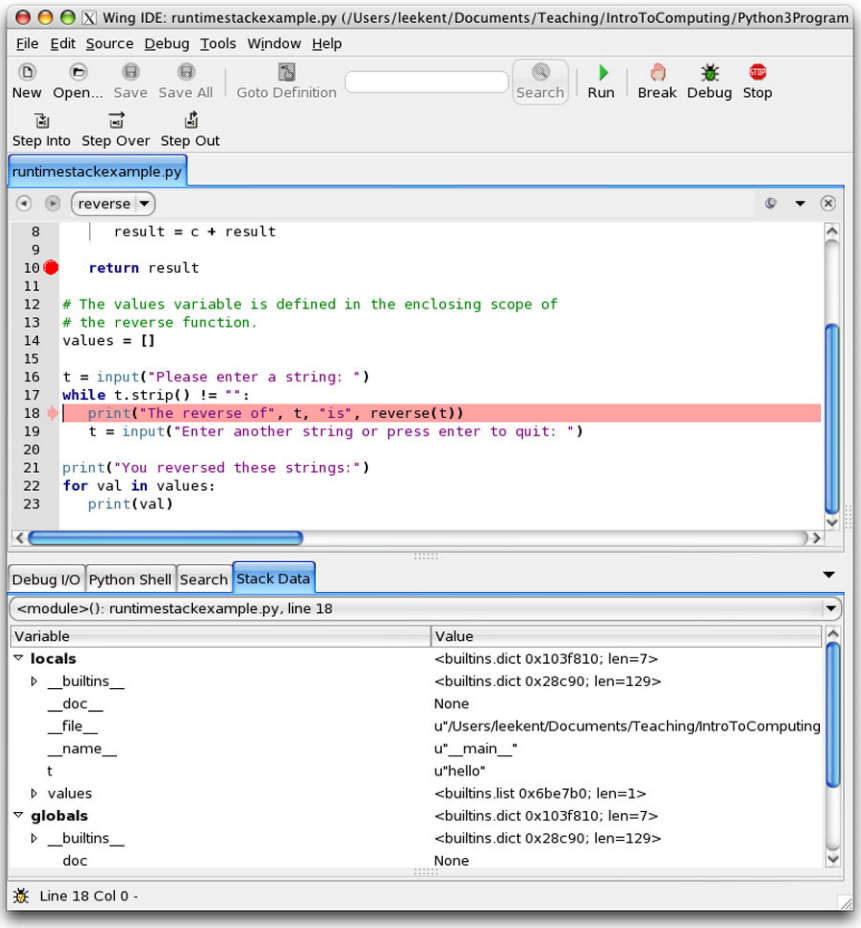

Fig. 5.2 The run-time stack in the Wing IDE

Practice 5.5 Trace the execution of the code in Example 5.2 on paper showing the contents of the run-time stack just before the function call returns.

The run-time stack is visible in most debuggers including the Wing IDE. To view the activation records on the run-time stack you have to debug your program and set a breakpoint during its execution. Figure 5.2 shows the Wing IDE running the program from Example 5.4. A breakpoint was set just before the reverse function returns: the same point at snapshot one in Fig. 5.1. In Wing you can click on the *Stack Data* tab to view the run-time stack. The drop-down combobox directly below the *Stack Data* tab contains one entry for

each activation record currently on the run-time stack. In Fig. 5.2 the <*module*> activation record is selected which is Wing's name for the *Main* activation record. When an activation record is selected in the *Stack Data* tab, its local variables are displayed below. In Fig. 5.2 the *t* and *values* variables are displayed from the *Main* activation record. The program is currently stopped at line 10 but the *reverse* function was called from line 18 so that line is highlighted since we are displaying the activation record corresponding to the code *reverse* was called from.

Practice 5.6 Trace the execution of the code in Example 5.2 using the Wing IDE to verify the contents of the run-time stack just before the function call returns match your answer in practice problem 5.5.

5.5 Mutable Data and Functions

If you consider Fig. 5.1, it should help in understanding that a function that mutates a value passed to it will cause the code that called it to see that mutated data. The program presented in Example 5.7 does not mutate any of the data passed to the reverse function. In fact, since strings are immutable, it would be impossible for reverse to mutate the parameter passed to it. However, Example 5.4 mutates the *values* list. The result of appending to the list is seen in the code that called it. Lists are not immutable. They can be changed in place. In Example 5.4 the reference to the *values* list is not changed. The *contents* of the values list is changed. The changed contents are seen by the main code after the function returns.

As another example, consider a reverse function that doesn't return a value. What if it just changed the list that was given to it. If the parameter to the list function was called *lst*, then writing *lst*[0] = "*h*" will change the first element of the list *lst* to the string *h*. That's what is meant by mutating a data. A new list is not created in this case. The existing list is modified. If a list is passed to a function and the function mutates the list, the caller of the function will see the reversed list. That's what the *append* method does. It mutates the existing list as well.

When a function is called that mutates one or more of its parameters, the calling code will see that the data has been mutated. The mutation is not somehow *undone* when the function returns. Since strings, ints, floats, and bools are all immutable, this never comes up when passing arguments of these types. But, again, lists are mutable, and a function may mutate a list as seen in the next example.

5

Example 5.8 Consider the following code that reverses a list in place. It does not build a new list. It reverses the existing list.

```
def reverseInPlace(lst):
    for i in range(len(lst)//2):
        tmp = lst[i]
        lst[i] = lst[len(lst)-1-i]
        lst[len(lst)-1-i]=tmp

s = input("Please enter a sentence:")
lst = s.split()
reverseInPlace(lst)
print("The sentence backwards is:",end="")
for word in lst:
    print(word,end=" ")
print()
```

Notice that the *reverseInPlace* function in Example 5.8 does not return anything. In addition, when *reverseInPlace* is called it is not set to some variable, nor is the return value printed. It is just called on a line by itself. That's because it modifies the list passed to it as an argument.

Practice 5.7 Why would it be very uninteresting to call *reverseInPlace* like this? What would the next line of code be?

```
print(reverseInPlace([1,2,3,4,5]))
```

In practice problem 5.7 the value printed to the screen is *None*. None is a special value in Python. It is returned by any function that does not *explicitly* return a value. All functions return a value in Python. Those that don't have a *return* statement in them to explicitly return a value, return *None* by default. Obviously, printing *None* wouldn't tell us much about the reverse of [*1*, *2*, *3*, *4*, *5*].

Practice 5.8 What would happen if you tried to use reverseInPlace to reverse a string?

5.6 Predicate Functions

A predicate is an answer to a question with respect to one or more objects. For instance, we can ask, *Is x even?*. If the value that *x* refers to is even, the answer would be *Yes* or *True*. If *x* refers to something that is not even then the answer would be *False*. In Python, if we write a function that returns `True` or `False` depending on its parameters, that function is called a *Predicate* function. Predicate functions are usually implemented using the *Guess and Check* pattern. However, applying this pattern to a function can look a little different than the pattern we learned about in Chap. 2.

Example 5.9 Assume we want to write a predicate function that returns True if one number evenly divides another and false otherwise. Here is one version of the code that looks like the old *Guess and Check* pattern.

```
def evenlyDivides(x,y):
    # returns true if x evenly divides y

    dividesIt = False

    if y % x == 0:
        dividesIt = True

    return dividesIt

x = int(input("Please enter an integer:"))
y = int(input("Please enter another integer:"))

if evenlyDivides(x,y):
    print(x,"evenly divides",y)
else:
    print(x,"does not evenly divide",y)
```

In Example 5.9 the guess and check pattern is applied to the function *evenlyDivides*. Observing that the function returns *True* or *False* it could be rewritten to just return that value instead of using a variable at all as in Example 5.10 below.

Example 5.10 In this example the value is just returned instead of storing it in a variable and returning at the bottom. This is equivalent to the code in Example 5.9 because it returns *True* and *False* in exactly the same instances as the other version of the function. NOTE: If *y % x == 0* then the *return True* is executed. This terminates the function immediately and it never gets to the statement *return False* in that case. If *y % x == 0* is false, then the code skips the *then* part of the *if* statement and executes the *return False*.

```
def evenlyDivides(x,y):
    # returns true if x evenly divides y
    if y % x == 0:
        return True

    return False

x = int(input("Please enter an integer:"))
y = int(input("Please enter another integer:"))

if evenlyDivides(x,y):
    print(x,"evenly divides",y)
else:
    print(x,"does not evenly divide",y)
```

Since the function in Examples 5.9 and 5.10 returns *True* when *y % x == 0* and *False* when it does not, there is one more version of this function that is even more concise in its definition. Any time you have an if statement where you see *if c is true then return true else return false* it can be replaced by *return c*. You don't need an if statement if all you want to do is return true or false based on one condition.

Example 5.11 Here is the same program one more time. This is the elegant version.

```
def evenlyDivides(x,y):
    return y % x == 0

x = int(input("Please enter an integer:"))
y = int(input("Please enter another integer:"))

if evenlyDivides(x,y):
    print(x,"evenly divides",y)
else:
    print(x,"does not evenly divide",y)
```

While the third version of the *evenlyDivides* function is the most elegant, this pattern may only be applied to predicate functions where only one condition needs to be checked. If we were trying to return write a predicate function that needed to check multiple conditions, then the second or first form of *evenlyDivides* would be required.

Practice 5.9 Write a function called *evenlyDividesList* that returns true if every element of a list given to the function is evenly divided by an integer given to the function.

5.7 Top-Down Design

Functions may be called from either the main code of a program or from other functions. A function call is allowed any place an expression may be written in Python. One technique for dealing with the complexity of writing a complex program is called *Top-Down Design*. In top-down design the programmer decides what major actions the program must take and then rather than worry about the details of how it is done, the programmer just defines a function that will handle that later.

Example 5.12 Assume we want to implement a program that will ask the users to enter a list of integers and then will answer which pairs of integers are evenly divisible. For instance, assume that the list of integers 1, 2, 3, 4, 5, 6, 8, and 12 were entered. The program should respond:

```
1 is evenly divisible by 1
2 is evenly divisible by 1 2
3 is evenly divisible by 1 3
4 is evenly divisible by 1 2 4
5 is evenly divisible by 1 5
6 is evenly divisible by 1 2 3 6
8 is evenly divisible by 1 2 4 8
12 is evenly divisible by 1 2 3 4 6 12
```

To accomplish this, a top down approach would start with getting the input from the user.

```
s = input("Please enter a list of ints separated by spaces:")
lst = []
for x in s.split():
    lst.append(int(x))

evenlyDivisible(lst)
```

Without worrying further about how *evenlyDivisible* works we can just assume that it will work once we get around to defining it. Of course, the program won't run until we define *evenlyDivisible*. But we *can* decide that *evenlyDivisible* must print a report to the screen the way the output is specified in Example 5.12. Later we can write the *evenlyDivisible* function. In a top-down design, when we write the *evenlyDivisible* function we would look to see if we could somehow make the job simpler by calling another function to help with the implementation. The *evenlyDivides* function could then be defined. In this way the main code calls a function to help with its implementation. Likewise, the *evenlyDivisible* function calls a function to aid in its implementation. This top-down approach continues until simple functions with straightforward implementations are all that is left.

5.8 Bottom-Up Design

In a *Bottom-Up Design* we would start by defining a simple function that might be useful in solving a more complex problem. For instance, the *evenlyDivides* function that checks to see if one value evenly divides another, could be useful in solving the problem presented in Example 5.12. Using a bottom-up approach a programmer would then see that *evenlyDivides* solves a slightly simpler problem and would look for a way to apply the *evenlyDivides* function to the problem we are solving.

Practice 5.10 Using the last version of the *evenlyDivides* function, write a function called *evenlyDivisibleElements* that given an integer, x, and a list of integers, returns the list of integers from the given list that evenly divide x. This would be the next step in either the bottom-up design or the top-down design of a solution to the problem in Example 5.12.

Practice 5.11 Write the function *evenlyDivisible* from Example 5.12 using the *evenlyDivisibleElements* function to complete the program presented in Example 5.12 and practice problem 5.10.

5.9 Recursive Functions

Sections 5.7 and 5.8 taught us that functions can call other functions and that sometimes this helps make a complex problem more manageable in some way. It turns out that not only can functions call other functions, they can also call themselves. This too can make a problem more manageable. If you've ever seen a proof by induction in Mathematics, recursive functions are somewhat like inductive proofs. In an inductive proof we are given a problem and told we know it is solvable for a smaller sized problem. Induction says that if we can use that smaller solution to arrive at a bigger solution, then we can conclude every instance of that problem has a solution. What makes an inductive proof so powerful is that we don't have to worry about the existence of a solution to the smaller problem. It is guaranteed to exist by the nature of the proof.

Recursion in functions works the same way. We may assume that our function will work if we call our function on a smaller value. Let's consider the computation of factorial from

Mathematics. $0! = 1$ by definition. This is called the base case. $n!$ is defined as $n \times (n-1)!$. This is the recursive part of the definition of factorial.

Example 5.13 Factorial can be written in Python much the same way it is defined in Mathematics. The *if* statement must come first and is the statement of the base case. The recursive case is always written last.

```
def factorial(n):
    if n == 0:
        return 1

    return n * factorial(n - 1)

print(factorial(5))
```

Practice 5.12 What would happen if the base case and the recursive case were written in the opposite order in Example 5.13? HINT: What happens to the run-time stack when a function is called?

A function is recursive if it calls itself. Recursion works in Python and other languages because of the run-time stack. To fully understand how the factorial function works, you need to examine the run-time stack to see how the program prints 120 to the screen.

Practice 5.13 Recalling that each time a function is called an activation record is pushed on the run-time stack, how many activation records will be pushed on the run-time stack at its deepest point when computing factorial(5)?

Practice 5.14 Run the factorial program on an input of 5 using Wing or your favorite IDE. Set a breakpoint in the factorial function on the two return statements. Watch the run-time stack grow and shrink. What do you notice about the parameter n?

Many problems can be formulated in terms of recursion. For instance, reversing a string can be formulated recursively. To reverse a string we only need to reverse a shorter string,

say all but the first letter, and then tack the first letter onto the other end of the reversed string. Here is the beautiful part of recursion. We can assume that reversing a shorter string already works!!!

Example 5.14 Here is a recursive version of a function that reverses a string. Remember, the base case must always come first. The base case usually defines the simplest problem we could come up with. The result of reversing an empty string is pretty easy to find. It is just the empty string.

```
def reverse(s):
    # Base Case: Always FIRST
    if s == "":
        return ""

    # Recursive Case: We may assume it works for
    # smaller problems. So, it works for the slice starting
    # at index 1 of the string.
    return reverse(s[1:]) + s[0]

print(reverse("hello"))
```

Practice 5.15 Write a recursive function that computes the nth Fibonacci number. The Fibonacci numbers are defined as follows: $\text{Fib}(0) = 1$, $\text{Fib}(1) = 1$, $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$. Write this as a Python function and then write some code to find the tenth Fibonacci number.

5.10 The Main Function

In most programming languages one special function is identified as the *main* function. The main function is where everything gets started. When a program in Java runs, the main function is executed first and the code in the main function determines what the program does. The same is true in C, C++, Pascal, Fortran, and many other languages. In Python this is not required by the language. However, it is good programming practice to have a main function anyway.

One advantage to defining a main function is when you wish to write a module that others may use. When importing a module a programmer probably does not want the main function in the imported module to run since he or she is undoubtably writing their own main function. The programmer writing the module that is imported may want to write a

main function to test the code they are providing in the module. Python has some special handling of imported modules that allow both the provider and the importer of a module to get the behavior they desire.

By writing a main function, all variables defined in the main function are no longer available to the whole program module. An example might help in explaining why this might be important.

Example 5.15 This code works, but it is accessing the variable *l* in the *drawSquare* function from the enclosing scope. It is generally a bad idea to access the enclosing scope of a function except in some specific circumstances. Of course, this was a mistake. It should have been *length* that was used in the *drawSquare* function.

```
# Imports always go at the top
import turtle

# Function definitions go second
def drawSquare(turtle, length):
    for k in range(4):
        turtle.forward(l)
        turtle.left(90)

# Main code goes at the end
t = turtle.Turtle()
screen = t.getscreen()

l = int(input("Please enter a side length: "))
drawSquare(t,l)

screen.exitonclick()
```

While the code in Example 5.15 works, it is not desirable because if a programmer changes the main code he or she may affect the code in the drawSquare function. For instance, if the programmer renames *l* to *length* at some future time, then the drawSquare function will cease to work. In addition, if drawSquare is moved to another module at some point in the future it will cease to work. A function should be as self-contained as possible to make it independent of where it is defined and where it is used.

The problem in the code above is easy to miss at first. You could easily think the program is fine since it does what it is supposed to do. The problem is due to the fact that up to this point we have not used a main function in our programs. Python programmers sometimes write a main function and sometimes do not. However, it is safer to write a main function and most experienced Python programmers will stick to the convention of writing one.

5

Example 5.16 Here is the draw square program again, this time with a main function. When the Python interpreter scans this file, two functions are defined, *drawSquare* and *main*. The *if* statement at the end of the program is the first statement to be executed.

```
# Imports always go at the top
import turtle

# Function definitions go second
def drawSquare(turtle, length):
    for k in range(4):
        turtle.forward(length)
        turtle.left(90)

# main function definition goes second to last.
def main():
    t = turtle.Turtle()
    screen = t.getscreen()
    l = int(input("Please enter a side length: "))
    drawSquare(t,l)
    screen.exitonclick()

# the if statement that calls main goes last.
if __name__ == "__main__":
    main()
```

When a program has a main function in Python, the convention is to write an *if* statement at the end of the program that starts everything executing. There is a special hook in Python that controls how a Python program is started. When a program is imported as a module the special variable called *__name__* is set to the name of the module. When a program is NOT imported, but run as the main module of a Python program, the special variable *__name__* is set to the value "*__main__*". When running the code in Example 5.16 the *if* statement's condition is True and therefore *main* is called to get the program started. However, this code implements a useful function, the *drawSquare* function. It might be the case that some programmer would like to use this function in their code. If this code resides a file called *square.py* and a programmer has a copy of this module and writes *import square* in their code, then when this module loads the *__name__* variable will be set to the name of the module and not "*__main__*". If you run this code as a program then the *main* function gets called. If you import this module into some other program, then the *main* function does not get called. When a module is written that is intended to be imported into other code, the main function often contains code to test the functions provided in the module.

In Example 5.16, if the programmer were to mistakenly write *turtle.forward*(*l*) instead of *turtle.forward*(*length*), Python would complain the first time the *drawSquare* function was called. It would say that *l* is undefined. This is much more desirable since we would like to catch errors like that right away as opposed to some later time.

Example 5.17 Here are a few lines from the *turtle.py* module that would be executed when the turtle module is run as a program instead of being imported.

```
if __name__ == "__main__":
    def switchpen():
        if isdown():
            pu()
        else:
            pd()

    def demo1():
        """Demo of old turtle.py - module"""
        reset()
        ...

    def demo2():
        """Demo of some new features."""
        speed(1)
        ...

    demo1()
    demo2()
    exitonclick()
```

5.11 Keyword Arguments

Up to this point we have learned that arguments passed to a function must be in the same order as the formal parameters in the function definition. For instance, in Example 5.16, to call the *drawSquare* function we would write *drawSquare*(*t*,*l*) as is done in the *main* function of the example.

It turns out that Python allows programmers to call functions using keyword arguments as well [5]. This is not possible in every language, but this is one of the very powerful features of Python. A formal parameter in the function definition is the name given to a value that will be passed to the function. For instance, in Example 5.16 the formal parameters to drawSquare are *turtle* and *length*. These two names are also keywords that may be used when calling drawSquare. The drawSquare function can be called by writing *drawSquare*(*length*=*l*,*turtle*=*t*) using the keyword style of parameter passing.

5.12 Default Values

When the keyword style of parameter passing is used, some keyword values may or may not be supplied depending on what the function does. In this case, a function definition can supply a default value for a parameter.

Example 5.18 Here is the *drawSquare* function with a default length value for the side length of the square. This means that the following calls to *drawSquare* would all be valid.

```
def drawSquare(turtle,length=20):
    for k in range(4):
        turtle.forward(length)
        turtle.left(90)

drawSquare(t,40)
drawSquare(t)
drawSquare(length=30,turtle=t)
```

5.13 Functions with Variable Number of Parameters

Python functions may have a variable number of parameters passed to them. To deal with this a special form of parameter is defined in Python by writing an asterisk in front of it. Writing **args* as a formal parameter defines *args* as a list (see [5]). Every argument that is passed starting at *args* position will be passed in a list that *args* will refer to.

Example 5.19 Consider a function called *drawFigure* that draws a figure by making a series of forward and left moves with a turtle. Since there could be a variable number of forward and left turns, they are represented by the formal parameter **args* which is a list of all the arguments after the named *turtle* argument.

```
import turtle

def drawFigure(turtle, *args):
    for i in range(0,len(args),2):
        turtle.forward(args[i])
        turtle.left(args[i+1])

def main():
    t = turtle.Turtle()
    screen = t.getscreen()
    drawFigure(t,50,90,30,90,50,90,30,90)
    screen.exitonclick()

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

5.14 Dictionary Parameter Passing

Using keyword/value pairs to pass values to functions is much like building a dictionary. A dictionary is a set of keys and associated values. For instance, you can assign `width=20` and `height=40` in a dictionary. Appendix E describes the operators and methods of dictionaries.

Example 5.20 Here is a dictionary called *dimensions* with keys *width* and *height*.

```
dimensions = {}
dimensions['width'] = 20
dimensions['height'] = 40
```

As an added convenience for programmers, a dictionary of keyword/value pairs may be specified as a parameter to a function [5]. The dictionary is automatically defined as the set of all keyword/value pairs passed to the function. A keyword/value dictionary parameter is defined by writing two asterisks in front of the parameter name.

Example 5.21 Here is a *drawRectangle* function that gets its width and height as keyword/-value arguments. The function definition specifies a *dimensions* keyword/value dictionary argument. The code below shows how it can be used.

```
import turtle

def drawRectangle(turtle, **dimensions):
    width = 10
    height = 10
    if "width" in dimensions:
        width = dimensions["width"]
    if "height" in dimensions:
        height = dimensions["height"]
    drawFigure(turtle,width,90,height,90,width,90,height,90)

def drawFigure(turtle, *args):
    for i in range(0,len(args),2):
        turtle.forward(args[i])
        turtle.left(args[i+1])

def main():
    t = turtle.Turtle()
    screen = t.getscreen()
    drawRectangle(t,width=40,height=20)
    screen.exitonclick()

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

5.15 Review Questions

1. What is the difference between defining a function and calling a function? Give an example of each and describe what happens when a function is both defined and called.
2. What are two reasons to write functions when possible in your code?
3. What is an argument and what is a formal parameter?
4. What is scope and what is the name of the rule for determining the scope of a variable? Describe what each letter means in the acronym for determining scope.
5. What is an activation record? When is one pushed and when is it popped?
6. How do activation records and scope relate to each other?
7. If a function is called and passed a string it can make all the changes it wants to the string but when the function returns the changes will be lost. This isn't necessarily the case if a function is passed a list. Why?
8. What is a predicate function? What programming pattern is a predicate function likely going to use?
9. What is the difference between top-down and bottom-up design?
10. What is a recursive function? What two things must a recursive function contain?
11. Why is a *main* function beneficial in a program? Give two reasons a main function might help in the implementation of a module.
12. What is a keyword parameter/argument? How does it differ from a regular argument?
13. What is a dictionary? How can a dictionary be used in parameter passing?

5.16 Exercises

1. Write a program that contains a drawTruck function that given an x, y coordinate on the screen draws a truck using Turtle graphics. You may use the `goto` method on the first line of the function, but after that use only left, right, forward, and back to draw the truck. You may use color when drawing if you would like to.
2. Modify the program in the previous exercise to add a *scale* parameter to the drawTruck function. You should multiply the scale times each forward or back method call while drawing the truck. Then use the drawTruck function at least three times in a program to draw trucks of different sizes.
3. Write a program that contains a function called drawRegularPolygon where you give it a Turtle, the number of sides of the polygon, and the side length and it draws the polygon for you. NOTE: This function won't return a value since it has a side-effect of drawing the regular polygon. Then write some code that uses this function at least three times to draw polygons of different sizes and shapes.
4. Write a predicate function called isEven that returns True if a number is even and False if it is not. Use the function in a program and test your code on several different values.
5. Write a function called allEvens that given a list of integers, returns a new list containing only the even integers. Use the function in a program and test your code on several different values.

6. Write a function called isPalindrome that returns True if a string given to it is a palindrome. A palindrome is a string that is the same spelled backwards or forwards. For instance, *radar* is a palindrome. Use the function in a program and test your code on several different values.
7. Write a function called isPrime that returns True if an integer given to the function is a prime number. Use the function in a program and test your code on several different values.
8. A tuple is a sequence of comma separated values inside of parens. For instance (5,6) is a two-tuple. Write a function called *zip* that is given two lists of the same length and creates a new list of two-tuples where each two-tuple is the tuple of the corresponding elements from the two lists. For example, zip([1, 2, 3], [4, 5, 6]) would return [(1, 4), (2, 5), (3, 6)]. Use the function in a program and test your code on several different values.
9. Write a function called *unzip* that returns a tuple of two lists that result from unzipping a zipped list (see the previous exercise). So unzip([(1, 4), (2, 5), (3, 6)]) would return ([1, 2, 3], [4, 5, 6]). Use the function in a program and test your code on several different values.
10. Write a function called sumIt which is given a list of numbers and returns the sum of those numbers. Use the function in a program and test your code on several different values.
11. Write a recursive function called recursiveSumIt which given a list of numbers, returns the sum of those numbers. Use the function in a program and test your code on several different values.
12. Use top-down design to write a program with three functions that capitalizes the first letter of each word in a sentence. For instance, if the user enters "hi there how are you" the program should print back to the screen "Hi There How Are You". Don't forget to define at least three functions using top-down design. Write comments to show what function you wrote first, followed by the second function you wrote, followed by the third function you wrote assuming you employed a top-down design.
13. Use bottom-up design to write a program with three functions that capitalizes the first letter of each word in a sentence. For instance, if the user enters "hi there how are you" the program should print back to the screen "Hi There How Are You". Don't forget to define at least three functions using bottom-up design. Write comments to show what function you wrote first, followed by the second function you wrote, followed by the third function you wrote assuming you employed a bottom-up design. HINT: The answer to this problem and exercise 12 should only differ in the order that you wrote the functions. The solutions should otherwise be identical.
14. Write a function called *factors* that given an integer returns the list of the factors of that integer. For instance, *factors*(6) would return [1, 2, 3, 6].
15. Write a function called *sumFactors* that given an integer returns the sum of the factors of that integer. For instance, *sumFactors*(6) would return 12 since $1 + 2 + 3 + 6 = 12$.
16. Write a function called *isPerfect* that given an integer returns True if the number is the sum of its factors (not including itself) and False otherwise. For instance, 6 is a perfect number because its factors, 1, 2, and 3 add up to 6.

17. Write a function called *sumRange* that given two integers returns the sum of all the integers between the two given integers inclusive. For instance, *sumRange*(3, 6) would return 18. Use a second function in the definition of sumRange to show that you can employ some top-down design to decompose this problem into a simpler problem and then use that simpler solution to solve this problem. HINT: Look for a function in these exercises you might use in defining *sumRange*.
18. Write a function called *reverseWords* that given a string representing a sentence, returns the same sentence but with each word reversed. For instance, *reverseWords*("*hi there how are you*") would return "ih ereht woh era uoy". Use another function in the definition of this function to make the task of writing this program simpler.
19. Write a function called *oddCharacters* that given a string, returns a string containing only the odd characters of the given string. The first element of a string (i.e. index 0) is an even element. *oddCharacters*("*hi there*") should be "*itee*".
20. Write a function called *oddElements* that given a list, returns a list containing only the odd elements of the list. The first element of a list (i.e. index 0) is an even element. *oddElements*([1,2,3,4]) should be [2,4]. What do you notice about this and the previous problem?
21. Write a function called *dotProduct* that computes the dot product of two lists of numbers given to the function. Use the *zip* function in your solution.
22. Review exercise 2 from Chap. 3. Use top-down design to write at least two functions that implement an addressbook application as described there. When you write it this time use the technique of parallel lists introduced in Chap. 4. The program should read all the records from the file and place the contents of the fields of each record in parallel lists so the file does not have to be read more than once in the application. But, be sure to write the contents of the parallel lists to the file when the user chooses to quit. Otherwise, you won't be able to add entries to the address book.
23. Write a program that computes a user's GPA on a 4 point scale.Each grade on a 4 point scale is multiplied by the number of credits for that class. The sum of all the credit, grade products is divided by the total number of credits earned. Assume the 4 point scale assigns values of 4.0 for an A, 3.7 for an A-, 3.3 for a B+, 3.0 for a B, 2.7 for a B-, 2.3 for a C+, 2.0 for a C, 1.7 for a C-, 1.3 for a D+, 1.0 for a D, 0.7 for a D-, and 0 for an F. Ask the user to enter their credit grade pairs using the following format until the enter 0 for the number of credits.

    In this version of the program you should read the data from the user and build parallel lists. Then, write a function called computeWeightedAverage that given the two parallel lists computes the average and returns it. Use this function in your program.

```
This program computes your GPA.
Please enter your completed courses.
Terminate your entry by entering 0 credits.
Credits? 4
Grade? A
Credits? 3
Grade? B+
Credits? 4
Grade? B-
```

```
Credits? 2
Grade? C
Credits? 0
Your GPA is 3.13
```

5.17 Solutions to Practice Problems

These are solutions to the practice problems in this chapter. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

Solution to Practice Problem 5.1

```
def explode(s):
  lst = []
  for c in s:
    lst.append(c)

  return lst

print(explode("hello"))
```

Solution to Practice Problem 5.2

```
def implode(lst):
  s = ""
  for e in lst:
    s = s+e

  return s

print(implode(['h', 'e', 'l', 'l', 'o']))
```

Solution to Practice Problem 5.3

The error is *variable referenced before assignment*. It occurs on the first line, the second occurrence of *x*. At this point *x* has no value.

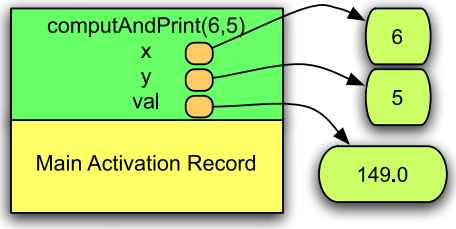

Fig. 5.3 The run-time stack for Example 5.2

Solution to Practice Problem 5.4

The error message is below. The problem is that the *len* function's name was overridden in the local scope by the *len* variable. This means that within the local scope of the *length* function, *len* cannot be called as a function. The error message says that an *int* is not callable.

```
Traceback (most recent call last):
  File "/Applications/WingIDE.app/...", line 8, in <module>
  File "/Applications/WingIDE.app/...", line 3, in length
    pass
builtins.TypeError: 'int' object is not callable
```

Solution to Practice Problem 5.5

Figure 5.3 show the contents of the run-time stack just before the return from the function. There are no variables in the main activation record.

Solution to Practice Problem 5.6

Refer to Fig. 5.3 to compare to what you see using your IDE.

Solution to Practice Problem 5.7

None is returned by the function since it does not explicitly return a value. So printing None is not very interesting, But, more importantly, since the list is reversed in place then how should the list be accessed? There is no reference stored to the list once the function returns so the garbage collector comes along and reclaims the space throwing away the work that was just done. The correct way to call it is shown in Example 5.8.

Solution to Practice Problem 5.8

The *reverseInPlace* function cannot be used to reverse a string since indexed assignment is not possible on strings. In other words, strings are immutable. The line of code $lst[i] = lst[len(lst) - 1 - i]$ is the line of code where the program would terminate abnormally.

Solution to Practice Problem 5.9

```
def evenlyDividesList(x,lst):

  for e in lst:
    if not evenlyDivides(x,e):
      return False

  return True
```

Solution to Practice Problem 5.10

```
def evenlyDivisibleElements(x,lst):
   result = []

   for e in lst:
     if evenlyDivides(e,x):
       result.append(e)

   return result
```

Solution to Practice Problem 5.11

```
def evenlyDivisible(lst):

  for e in lst:
    print(e,'is evenly divisible by ',end="")
    elements = evenlyDivisibleElements(e,lst)
    for f in elements:
       print(f,end=" ")

    print()
```

Solution to Practice Problem 5.12

Each time a function call is made an activation record is pushed on the stack. Each activation record takes some space. Without the base case first, the program would repeatedly call the *factorial* function until the run-time stack overflowed (i.e. ran out of space). This is called *infinite recursion* even though it will not continue indefinitely.

Solution to Practice Problem 5.13

There would be 7 activation records at its deepest point, one for the *main* activation record, and one for each of the arguments recursively passed to *factorial*(5): 5, 4, 3, 2, 1, 0.

Solution to Practice Problem 5.14

When you run the program you should notice that there are 6 different *n* variables, each with a different value from 5 to 0. This is why it is important to understand the run-time stack and how it works when dealing with recursion. Recursive functions cannot work without the run-time stack.

Solution to Practice Problem 5.15

Here is the solution. However, you would never, ever, write such a program and use it in a commercial setting. It is too slow for anything but small values of *n*. There are much better solutions to finding fibonacci numbers that are available.

```
def recfib(n):
    if n == 0:
        return 1

    if n == 1:
        return 1

    return fib(n-1) + fib(n-2)
```