1.5 Functions

In this section, we explore the creation of and use of functions in Python. As we did in Section 1.2.2, we draw a distinction between ***functions*** and ***methods***. We use the general term *function* to describe a traditional, stateless function that is invoked without the context of a particular class or an instance of that class, such as sorted(data). We use the more specific term *method* to describe a member function that is invoked upon a specific object using an object-oriented message passing syntax, such as data.sort( ). In this section, we only consider pure functions; methods will be explored with more general object-oriented principles in Chapter 2.

We begin with an example to demonstrate the syntax for defining functions in Python. The following function counts the number of occurrences of a given target value within any form of iterable data set.

```
def count(data, target):
  n = 0
  for item in data:
    if item == target:                    # found a match
      n += 1
  return n
```

The first line, beginning with the keyword **def**, serves as the function's ***signature***. This establishes a new identifier as the name of the function (count, in this example), and it establishes the number of parameters that it expects, as well as names identifying those parameters (data and target, in this example). Unlike Java and C++, Python is a dynamically typed language, and therefore a Python signature does not designate the types of those parameters, nor the type (if any) of a return value. Those expectations should be stated in the function's documentation (see Section 2.2.3) and can be enforced within the body of the function, but misuse of a function will only be detected at run-time.

The remainder of the function definition is known as the ***body*** of the function. As is the case with control structures in Python, the body of a function is typically expressed as an indented block of code. Each time a function is called, Python creates a dedicated ***activation record*** that stores information relevant to the current call. This activation record includes what is known as a ***namespace*** (see Section 1.10) to manage all identifiers that have ***local scope*** within the current call. The namespace includes the function's parameters and any other identifiers that are defined locally within the body of the function. An identifier in the local scope of the function caller has no relation to any identifier with the same name in the caller's scope (although identifiers in different scopes may be aliases to the same object). In our first example, the identifier n has scope that is local to the function call, as does the identifier item, which is established as the loop variable.

3.2 The Seven Functions Used in This Book

In this section, we briefly discuss the seven most important functions used in the analysis of algorithms. We will use only these seven simple functions for almost all the analysis we do in this book. In fact, a section that uses a function other than one of these seven will be marked with a star ($\star$) to indicate that it is optional. In addition to these seven fundamental functions, Appendix B contains a list of other useful mathematical facts that apply in the analysis of data structures and algorithms.

The Constant Function

The simplest function we can think of is the ***constant function***. This is the function,

$$f(n) = c,$$

for some fixed constant c, such as $c = 5$, $c = 27$, or $c = 2^{10}$. That is, for any argument n, the constant function $f(n)$ assigns the value c. In other words, it does not matter what the value of n is; $f(n)$ will always be equal to the constant value c.

Because we are most interested in integer functions, the most fundamental constant function is $g(n) = 1$, and this is the typical constant function we use in this book. Note that any other constant function, $f(n) = c$, can be written as a constant c times $g(n)$. That is, $f(n) = cg(n)$ in this case.

As simple as it is, the constant function is useful in algorithm analysis, because it characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers.

The Logarithm Function

One of the interesting and sometimes even surprising aspects of the analysis of data structures and algorithms is the ubiquitous presence of the ***logarithm function***, $f(n) = \log_b n$, for some constant $b > 1$. This function is defined as follows:

$$x = \log_b n \quad \text{if and only if} \quad b^x = n.$$

By definition, $\log_b 1 = 0$. The value b is known as the ***base*** of the logarithm.

The most common base for the logarithm function in computer science is 2, as computers store integers in binary, and because a common operation in many algorithms is to repeatedly divide an input in half. In fact, this base is so common that we will typically omit it from the notation when it is 2. That is, for us,

$$\log n = \log_2 n.$$

The Linear Function

Another simple yet important function is the ***linear function***,

$$f(n) = n.$$

That is, given an input value n, the linear function f assigns the value n itself.

This function arises in algorithm analysis any time we have to do a single basic operation for each of n elements. For example, comparing a number x to each element of a sequence of size n will require n comparisons. The linear function also represents the best running time we can hope to achieve for any algorithm that processes each of n objects that are not already in the computer's memory, because reading in the n objects already requires n operations.

The N-Log-N Function

The next function we discuss in this section is the ***n-log-n function***,

$$f(n) = n \log n,$$

that is, the function that assigns to an input n the value of n times the logarithm base-two of n. This function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function; therefore, we would greatly prefer an algorithm with a running time that is proportional to $n \log n$, than one with quadratic running time. We will see several important algorithms that exhibit a running time proportional to the n-log-n function. For example, the fastest possible algorithms for sorting n arbitrary values require time proportional to $n \log n$.

The Quadratic Function

Another function that appears often in algorithm analysis is the ***quadratic function***,

$$f(n) = n^2.$$

That is, given an input value n, the function f assigns the product of n with itself (in other words, "n squared").

The main reason why the quadratic function appears in the analysis of algorithms is that there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. Thus, in such cases, the algorithm performs $n \cdot n = n^2$ operations.

This function does not use any explicit loops. Repetition is provided by the repeated recursive invocations of the function. There is no circularity in this definition, because each time the function is invoked, its argument is smaller by one, and when a base case is reached, no further recursive calls are made.

We illustrate the execution of a recursive function using a ***recursion trace***. Each entry of the trace corresponds to a recursive call. Each new recursive function call is indicated by a downward arrow to a new invocation. When the function returns, an arrow showing this return is drawn and the return value may be indicated alongside this arrow. An example of such a trace for the factorial function is shown in Figure 4.1.

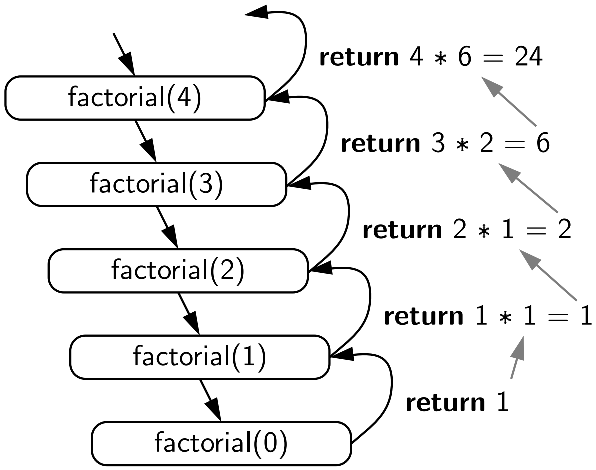

Figure 4.1: A recursion trace for the call factorial(5).

A recursion trace closely mirrors the programming language's execution of the recursion. In Python, each time a function (recursive or otherwise) is called, a structure known as an ***activation record*** or ***frame*** is created to store information about the progress of that invocation of the function. This activation record includes a namespace for storing the function call's parameters and local variables (see Section 1.10 for a discussion of namespaces), and information about which command in the body of the function is currently executing.

When the execution of a function leads to a nested function call, the execution of the former call is suspended and its activation record stores the place in the source code at which the flow of control should continue upon return of the nested call. This process is used both in the standard case of one function calling a different function, or in the recursive case in which a function invokes itself. The key point is that there is a different activation record for each active call.

15.1.3 Additional Memory Used by the Python Interpreter

We have discussed, in Section 15.1.1, how the Python interpreter allocates memory for objects within a memory heap. However, this is not the only memory that is used when executing a Python program. In this section, we discuss some other important uses of memory.

The Run-Time Call Stack

Stacks have a most important application to the run-time environment of Python programs. A running Python program has a private stack, known as the ***call stack*** or ***Python interpreter stack***, that is used to keep track of the nested sequence of currently active (that is, nonterminated) invocations of functions. Each entry of the stack is a structure known as an ***activation record*** or ***frame***, storing important information about an invocation of a function.

At the top of the call stack is the activation record of the ***running call***, that is, the function activation that currently has control of the execution. The remaining elements of the stack are activation records of the ***suspended calls***, that is, functions that have invoked another function and are currently waiting for that other function to return control when it terminates. The order of the elements in the stack corresponds to the chain of invocations of the currently active functions. When a new function is called, an activation record for that call is pushed onto the stack. When it terminates, its activation record is popped from the stack and the Python interpreter resumes the processing of the previously suspended call.

Each activation record includes a dictionary representing the local namespace for the function call. (See Sections 1.10 and 2.5 for further discussion of namespaces). The namespace maps identifiers, which serve as parameters and local variables, to object values, although the objects being referenced still reside in the memory heap. The activation record for a function call also includes a reference to the function definition itself, and a special variable, known as the ***program counter***, to maintain the address of the statement within the function that is currently executing. When one function returns control to another, the stored program counter for the suspended function allows the interpreter to properly continue execution of that function.

Implementing Recursion

One of the benefits of using a stack to implement the nesting of function calls is that it allows programs to use ***recursion***. That is, it allows a function to call itself, as discussed in Chapter 4. We implicitly described the concept of the call stack and the use of activation records within our portrayal of ***recursion traces*** in

Return Statement

A **return** statement is used within the body of a function to indicate that the function should immediately cease execution, and that an expressed value should be returned to the caller. If a return statement is executed without an explicit argument, the None value is automatically returned. Likewise, None will be returned if the flow of control ever reaches the end of a function body without having executed a return statement. Often, a return statement will be the final command within the body of the function, as was the case in our earlier example of a count function. However, there can be multiple return statements in the same function, with conditional logic controlling which such command is executed, if any. As a further example, consider the following function that tests if a value exists in a sequence.

```
def contains(data, target):
  for item in target:
    if item == target:                    # found a match
      return True
  return False
```

If the conditional within the loop body is ever satisfied, the return True statement is executed and the function immediately ends, with True designating that the target value was found. Conversely, if the for loop reaches its conclusion without ever finding the match, the final return False statement will be executed.

1.5.1 Information Passing

To be a successful programmer, one must have clear understanding of the mechanism in which a programming language passes information to and from a function. In the context of a function signature, the identifiers used to describe the expected parameters are known as ***formal parameters***, and the objects sent by the caller when invoking the function are the ***actual parameters***. Parameter passing in Python follows the semantics of the standard ***assignment statement***. When a function is invoked, each identifier that serves as a formal parameter is assigned, in the function's local scope, to the respective actual parameter that is provided by the caller of the function.

For example, consider the following call to our count function from page 23:

```
prizes = count(grades, 'A')
```

Just before the function body is executed, the actual parameters, grades and 'A', are implicitly assigned to the formal parameters, data and target, as follows:

```
data = grades
target = 'A'
```

1.12 Exercises

For help with exercises, please visit the site, www.wiley.com/college/goodrich.

Reinforcement

R-1.1 Write a short Python function, is_multiple(n, m), that takes two integer values and returns True if n is a multiple of m, that is, $n = mi$ for some integer i, and False otherwise.

R-1.2 Write a short Python function, is_even(k), that takes an integer value and returns True if k is even, and **False** otherwise. However, your function cannot use the multiplication, modulo, or division operators.

R-1.3 Write a short Python function, minmax(data), that takes a sequence of one or more numbers, and returns the smallest and largest numbers, in the form of a tuple of length two. Do not use the built-in functions min or max in implementing your solution.

R-1.4 Write a short Python function that takes a positive integer n and returns the sum of the squares of all the positive integers smaller than n.

R-1.5 Give a single command that computes the sum from Exercise R-1.4, relying on Python's comprehension syntax and the built-in sum function.

R-1.6 Write a short Python function that takes a positive integer n and returns the sum of the squares of all the odd positive integers smaller than n.

R-1.7 Give a single command that computes the sum from Exercise R-1.6, relying on Python's comprehension syntax and the built-in sum function.

R-1.8 Python allows negative integers to be used as indices into a sequence, such as a string. If string s has length n, and expression s[k] is used for index $-n \leq k < 0$, what is the equivalent index $j \geq 0$ such that s[j] references the same element?

R-1.9 What parameters should be sent to the range constructor, to produce a range with values 50, 60, 70, 80?

R-1.10 What parameters should be sent to the range constructor, to produce a range with values 8, 6, 4, 2, 0, -2, -4, -6, -8?

R-1.11 Demonstrate how to use Python's list comprehension syntax to produce the list [1, 2, 4, 8, 16, 32, 64, 128, 256].

R-1.12 Python's random module includes a function choice(data) that returns a random element from a non-empty sequence. The random module includes a more basic function randrange, with parameterization similar to the built-in range function, that return a random choice from the given range. Using only the randrange function, implement your own version of the choice function.

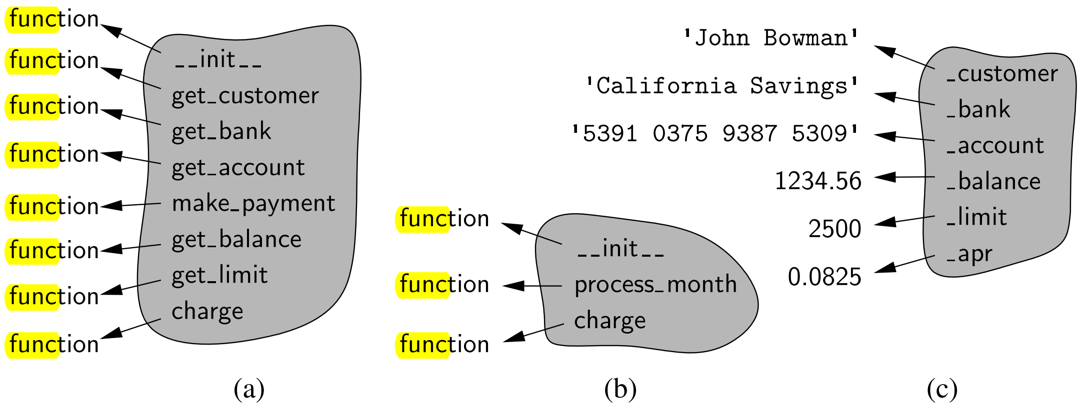

Figure 2.8: Conceptual view of three namespaces: (a) the class namespace for CreditCard; (b) the class namespace for PredatoryCreditCard; (c) the instance namespace for a PredatoryCreditCard object.

How Entries Are Established in a Namespace

It is important to understand why a member such as _balance resides in a credit card's instance namespace, while a member such as make_payment resides in the class namespace. The balance is established within the __init__ method when a new credit card instance is constructed. The original assignment uses the syntax, self._balance = 0, where self is an identifier for the newly constructed instance. The use of self as a qualifier for self._balance in such an assignment causes the _balance identifier to be added directly to the instance namespace.

When inheritance is used, there is still a single *instance namespace* per object. For example, when an instance of the PredatoryCreditCard class is constructed, the _apr attribute as well as attributes such as _balance and _limit all reside in that instance's namespace, because all are assigned using a qualified syntax, such as self._apr.

A *class namespace* includes all declarations that are made directly within the body of the class definition. For example, our CreditCard class definition included the following structure:

```
class CreditCard:
  def make_payment(self, amount):
    ...
```

Because the make_payment function is declared within the scope of the CreditCard class, that function becomes associated with the name make_payment within the CreditCard class namespace. Although member functions are the most typical types of entries that are declared in a class namespace, we next discuss how other types of data values, or even other classes can be declared within a class namespace.

These assignment statements establish identifier data as an alias for grades and target as a name for the string literal 'A'. (See Figure 1.7.)

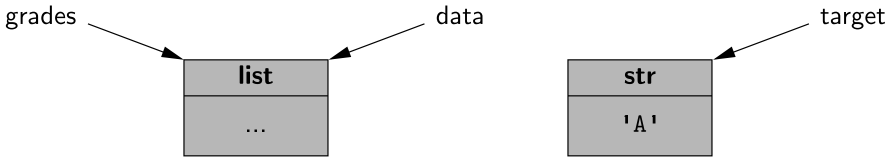

Figure 1.7: A portrayal of parameter passing in Python, for the function call count(grades, 'A'). Identifiers data and target are formal parameters defined within the local scope of the count function.

The communication of a return value from the function back to the caller is similarly implemented as an assignment. Therefore, with our sample invocation of prizes = count(grades, 'A'), the identifier prizes in the caller's scope is assigned to the object that is identified as n in the return statement within our function body.

An advantage to Python's mechanism for passing information to and from a function is that objects are not copied. This ensures that the invocation of a function is efficient, even in a case where a parameter or return value is a complex object.

Mutable Parameters

Python's parameter passing model has additional implications when a parameter is a mutable object. Because the formal parameter is an alias for the actual parameter, the body of the function may interact with the object in ways that change its state. Considering again our sample invocation of the count function, if the body of the function executes the command data.append('F'), the new entry is added to the end of the list identified as data within the function, which is one and the same as the list known to the caller as grades. As an aside, we note that reassigning a new value to a formal parameter with a function body, such as by setting data = [ ], does not alter the actual parameter; such a reassignment simply breaks the alias.

Our hypothetical example of a count method that appends a new element to a list lacks common sense. There is no reason to expect such a behavior, and it would be quite a poor design to have such an unexpected effect on the parameter. There are, however, many legitimate cases in which a function may be designed (and clearly documented) to modify the state of a parameter. As a concrete example, we present the following implementation of a method named scale that's primary purpose is to multiply all entries of a numeric data set by a given factor.

```
def scale(data, factor):
  for j in range(len(data)):
    data[j] *= factor
```

When an identifier is indicated in a command, Python searches a series of namespaces in the process of name resolution. First, the most locally enclosing scope is searched for a given name. If not found there, the next outer scope is searched, and so on. We will continue our examination of namespaces, in Section 2.5, when discussing Python's treatment of object-orientation. We will see that each object has its own namespace to store its attributes, and that classes each have a namespace as well.

First-Class Objects

In the terminology of programming languages, ***first-class objects*** are instances of a type that can be assigned to an identifier, passed as a parameter, or returned by a function. All of the data types we introduced in Section 1.2.3, such as int and list, are clearly first-class types in Python. In Python, functions and classes are also treated as first-class objects. For example, we could write the following:

```
scream = print          # assign name 'scream' to the function denoted as 'print'
scream('Hello')         # call that function
```

In this case, we have not created a new function, we have simply defined scream as an alias for the existing print function. While there is little motivation for precisely this example, it demonstrates the mechanism that is used by Python to allow one function to be passed as a parameter to another. On page 28, we noted that the built-in function, max, accepts an optional keyword parameter to specify a non-default order when computing a maximum. For example, a caller can use the syntax, max(a, b, key=abs), to determine which value has the larger absolute value. Within the body of that function, the formal parameter, key, is an identifier that will be assigned to the actual parameter, abs.

In terms of namespaces, an assignment such as scream = print, introduces the identifier, scream, into the current namespace, with its value being the object that represents the built-in function, print. The same mechanism is applied when a user-defined function is declared. For example, our count function from Section 1.5 beings with the following syntax:

```
def count(data, target):
    ...
```

Such a declaration introduces the identifier, count, into the current namespace, with the value being a function instance representing its implementation. In similar fashion, the name of a newly defined class is associated with a representation of that class as its value. (Class definitions will be introduced in the next chapter.)