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.

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.)

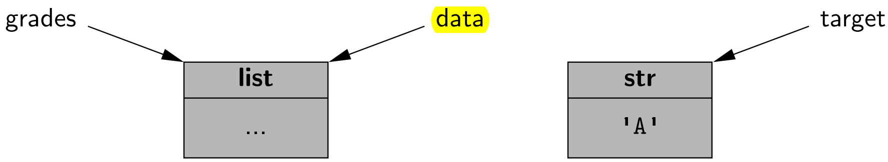

Figure 1.7: A portrayal of parameter passing in Python, for the function call count(grades, 'A'). Identifiers data and target are formal parameters defined within the local scope of the count function.

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
```

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,

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

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.

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.)

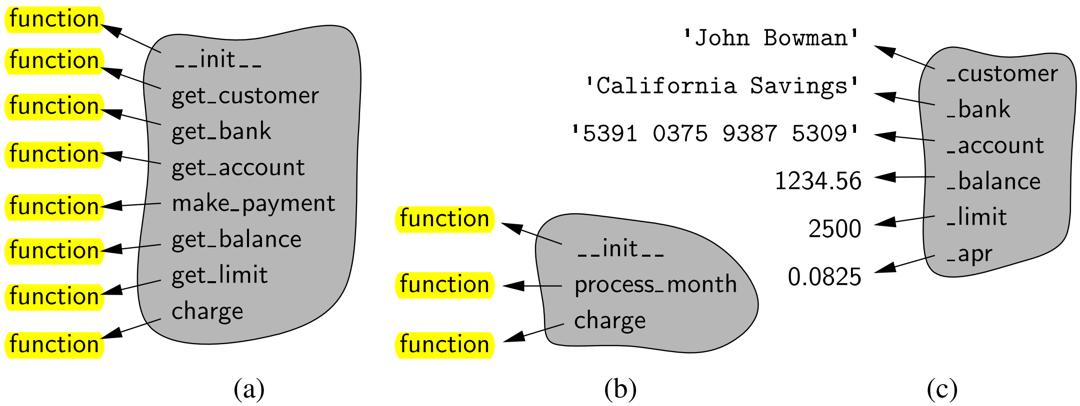

Figure 2.8: Conceptual view of three namespaces: (a) the class namespace for CreditCard; (b) the class namespace for PredatoryCreditCard; (c) the instance namespace for a PredatoryCreditCard object.

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.

Creativity

C-1.13 Write a pseudo-code description of a function that reverses a list of n integers, so that the numbers are listed in the opposite order than they were before, and compare this method to an equivalent Python function for doing the same thing.

C-1.14 Write a short Python function that takes a sequence of integer values and determines if there is a distinct pair of numbers in the sequence whose product is odd.

C-1.15 Write a Python function that takes a sequence of numbers and determines if all the numbers are different from each other (that is, they are distinct).

C-1.16 In our implementation of the scale function (page 25), the body of the loop executes the command data[j] *= factor. We have discussed that numeric types are immutable, and that use of the *= operator in this context causes the creation of a new instance (not the mutation of an existing instance). How is it still possible, then, that our implementation of scale changes the actual parameter sent by the caller?

C-1.17 Had we implemented the scale function (page 25) as follows, does it work properly?

```
def scale(data, factor):
  for val in data:
    val *= factor
```

Explain why or why not.

C-1.18 Demonstrate how to use Python's list comprehension syntax to produce the list [0, 2, 6, 12, 20, 30, 42, 56, 72, 90].

C-1.19 Demonstrate how to use Python's list comprehension syntax to produce the list ['a', 'b', 'c', ..., 'z'], but without having to type all 26 such characters literally.

C-1.20 Python's random module includes a function shuffle(data) that accepts a list of elements and randomly reorders the elements so that each possible order occurs with equal probability. The random module includes a more basic function randint(a, b) that returns a uniformly random integer from a to b (including both endpoints). Using only the randint function, implement your own version of the shuffle function.

C-1.21 Write a Python program that repeatedly reads lines from standard input until an EOFError is raised, and then outputs those lines in reverse order (a user can indicate end of input by typing ctrl-D).

Documentation

Python provides integrated support for embedding formal documentation directly in source code using a mechanism known as a ***docstring***. Formally, any string literal that appears as the *first* statement within the body of a module, class, or function (including a member function of a class) will be considered to be a docstring. By convention, those string literals should be delimited within triple quotes ("""). As an example, our version of the scale function from page 25 could be documented as follows:

```
def scale(data, factor):
  """Multiply all entries of numeric data list by the given factor."""
  for j in range(len(data)):
    data[j] *= factor
```

It is common to use the triple-quoted string delimiter for a docstring, even when the string fits on a single line, as in the above example. More detailed docstrings should begin with a single line that summarizes the purpose, followed by a blank line, and then further details. For example, we might more clearly document the scale function as follows:

```
def scale(data, factor):
  """Multiply all entries of numeric data list by the given factor.

  data       an instance of any mutable sequence type (such as a list)
             containing numeric elements

  factor     a number that serves as the multiplicative factor for scaling
  """
  for j in range(len(data)):
    data[j] *= factor
```

A docstring is stored as a field of the module, function, or class in which it is declared. It serves as documentation and can be retrieved in a variety of ways. For example, the command help(x), within the Python interpreter, produces the documentation associated with the identified object x. An external tool named pydoc is distributed with Python and can be used to generate formal documentation as text or as a Web page. Guidelines for *authoring* useful docstrings are available at:

http://www.python.org/dev/peps/pep-0257/

In this book, we will try to present docstrings when space allows. Omitted docstrings can be found in the online version of our source code.

By default, max operates based upon the natural order of elements according to the $<$ operator for that type. But the maximum can be computed by comparing some other aspect of the elements. This is done by providing an auxiliary *function* that converts a natural element to some other value for the sake of comparison. For example, if we are interested in finding a numeric value with *magnitude* that is maximal (i.e., considering -35 to be larger than $+20$), we can use the calling syntax max(a, b, key=abs). In this case, the built-in abs function is itself sent as the value associated with the keyword parameter key. (Functions are first-class objects in Python; see Section 1.10.) When max is called in this way, it will compare abs(a) to abs(b), rather than a to b. The motivation for the keyword syntax as an alternate to positional arguments is important in the case of max. This function is polymorphic in the number of arguments, allowing a call such as max(a,b,c,d); therefore, it is not possible to designate a key function as a traditional positional element. Sorting functions in Python also support a similar key parameter for indicating a nonstandard order. (We explore this further in Section 9.4 and in Section 12.6.1, when discussing sorting algorithms).

1.5.2 Python's Built-In Functions

Table 1.4 provides an overview of common functions that are automatically available in Python, including the previously discussed abs, max, and range. When choosing names for the parameters, we use identifiers x, y, z for arbitrary numeric types, k for an integer, and a, b, and c for arbitrary comparable types. We use the identifier, iterable, to represent an instance of any iterable type (e.g., str, list, tuple, set, dict); we will discuss iterators and iterable data types in Section 1.8. A sequence represents a more narrow category of indexable classes, including str, list, and tuple, but neither set nor dict. Most of the entries in Table 1.4 can be categorized according to their functionality as follows:

Input/Output: print, input, and open will be more fully explained in Section 1.6.

Character Encoding: ord and chr relate characters and their integer code points. For example, ord('A') is 65 and chr(65) is 'A'.

Mathematics: abs, divmod, pow, round, and sum provide common mathematical functionality; an additional math module will be introduced in Section 1.11.

Ordering: max and min apply to any data type that supports a notion of comparison, or to any collection of such values. Likewise, sorted can be used to produce an ordered list of elements drawn from any existing collection.

Collections/Iterations: range generates a new sequence of numbers; len reports the length of any existing collection; functions reversed, all, any, and map operate on arbitrary iterations as well; iter and next provide a general framework for iteration through elements of a collection, and are discussed in Section 1.8.